HOCHSCHULE RHEIN-WAAL
&
FLUXANA GMBH & CO. KG

BACHELOR'S THESIS

Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)

by investigating the influence of vibration, angle of dosing, and rotational speed to the mass flow of the powder.

Author:
Abdelrahman MOSTAFA
Matriculation No. 29528

Supervisor:
Prof. Dr. Ronny HARTANTO
Dr. Rainer SCHRAMM

A thesis submitted in fulfillment of the requirements for the Bachelor degree of Science

in the

Mechatronic Systems Engineering
Faculty of Technology & Bionics

November 8, 2023

Declaration of Authorship

I, Abdelrahman MOSTAFA, declare that this thesis titled, “Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

HOCHSCHULE RHEIN-WAAL

Abstract

Faculty of Technology & Bionics
Research & Development at Fluxana GmbH & Co. KG

Bachelor of Science

Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)

by Abdelrahman MOSTAFA

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor…

Contents

List of Figures

List of Tables

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
BASH	**B**ourn **A**gain **SH**ell
Cobot	**C**ollaborative R**obot**
DoF	**D**egree **o**f **F**reedom
FK	**F**orward **K**inematics
GUI	**G**rafical **U**ser **I**nterface
IK	**I**nverse **K**inematics
KDL	**K**inematics and **D**ynamics **L**ibrary
OpenCV	**Open** Source **C**omputer **V**ision Library
ROS	**R**obot **O**perating **S**ystem
TCP	**T**ool **C**enter **P**oint
TCP/IP	**T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol
TF	**T**rans**F**ormation
URDF	**U**nified **R**obot **D**escription **F**ormat

Chapter 1

Introduction

1.1 Motivation

1.2 Objectives

1.3 Thesis Structure

Part I

Basics of Robotics, ROS, and Powder Dosage

Chapter 2

Basics of Robotics

As an academic field, robotics emerges as a relatively youthful discipline, characterized by profoundly ambitious objectives, the most paramount of which is the creation of machines capable of emulating human behavior and cognitive processes. This quest to engineer intelligent machines inherently compels us to embark on a journey of self-exploration. It prompts us to scrutinize the intricacies of our own design—why our bodies possess the configurations they do, how our limbs synchronize in movement, and the mechanisms behind our acquisition and execution of intricate tasks. The realization that the fundamental inquiries in robotics are intrinsically linked to inquiries about our own existence forms a captivating and immersive aspect of the robotics pursuit. [20]

Definition

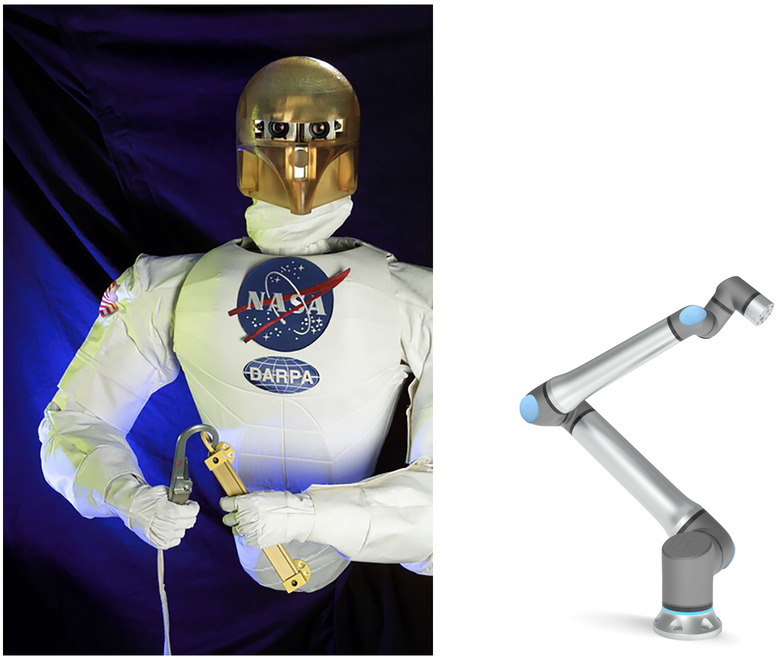
> **Robotics** is the scientific field dedicated to the study of robots—machines capable of autonomous operation, carrying out various tasks without direct human intervention. [19]

While science fiction often envisions robots in humanoid or android forms, real-world robots, especially those designed for industrial applications, typically deviate from human physical resemblance. These robots typically comprise three fundamental components: a mechanical structure, often represented by a robotic arm, enabling physical interaction with the robot's environment or itself; sensors that collect data on various physical attributes such as sound, temperature, motion, and pressure; and a processing system that interprets data from the robot's sensors, providing instructions for task execution.

It's worth noting that certain devices, like web-crawling search engine bots that systematically explore the internet to collect information on links and online content, may lack physical mechanical elements. Nonetheless, they are still classified as robots because they exhibit the ability to perform repetitive tasks autonomously.

This chapter delves into an exploration of various robot classifications, delving into the foundational principles of mechanics and kinematics. It also scrutinizes the intricacies of planning and control within the context of collaborative robots (cobots).

The knowledge presented in this chapter draws significant inspiration from two primary sources, namely '**Modern Robotics**' [20] by Kevin M. Lynch and Frank C. Park, and '**Theory of Applied Robotics**' [13] authored by Professor Reza N. Jazar. For a deeper understanding of these topics, I encourage you to refer to these texts.



(a) NASA Robonaut [1] [3] (b) Universal Robot (UR20) [41]

FIGURE 2.1: Figure (a) illustrates an instance of a humanoid robot developed by NASA, while Figure (b) exemplifies a cobot.

2.1 Types of Robots

Across diverse industries, robotics solutions have emerged as catalysts for heightened productivity, elevated safety standards, and increased operational adaptability. Organizations at the vanguard of innovation are discerning forward-looking applications of robotics that yield palpable and quantifiable outcomes. Intel collaborates closely with manufacturers, system integrators, and end-users, actively contributing to the realization of robots that deliver impactful, human-centered results.

According to an article from Intel regarding the classification of robots [12], the current generation of robots has been categorized into six distinct groups.

Autonomous Mobile Robots (AMRs) [8] AMRs navigate their environments and make rapid decisions on the fly. These robots employ advanced technologies like sensors and cameras to gather data from their surroundings. Equipped with onboard processing capabilities, they analyze this data and make well-informed decisions—whether it involves avoiding an approaching human worker, selecting the exact parcel to pick, or determining the suitable surface for disinfection. These robots are self-sufficient mobile solutions that operate with minimal human intervention. [32]

Automated Guided Vehicles (AGVs) [44] While AMRs navigate their surroundings autonomously, AGVs typically operate along fixed tracks or predetermined paths and frequently necessitate human supervision. AGVs find extensive application in scenarios involving the transportation of materials and goods within controlled settings like warehouses and manufacturing facilities.

Humanoids [17] While numerous mobile humanoid robots could, in a technical sense, be classified as Autonomous Mobile Robots (AMRs), this categorization primarily applies to robots fulfilling human-centric roles, frequently adopting human-like appearances. These robots leverage a similar array of technological components as AMRs to perceive, strategize, and execute tasks, encompassing activities such as offering navigational assistance or providing concierge services.

Hybrids [35] Diverse categories of robots are frequently integrated to engineer hybrid solutions that possess the capacity to execute intricate operations. For instance, the fusion of an AMR with a robotic arm can yield a versatile system tailored for the handling of packages within a warehouse environment. As functionalities are amalgamated within single solutions, there is a concurrent consolidation of computational capabilities.

Articulated Robots [34] Commonly referred to as robotic arms, are designed to replicate the versatile functions of the human arm. These systems typically incorporate a range of rotary joints, varying from two to as many as ten. The inclusion of additional joints or axes equips these robotic arms with a wider range of motion capabilities, rendering them particularly well-suited for tasks such as arc welding, material manipulation, machine operation, and packaging.

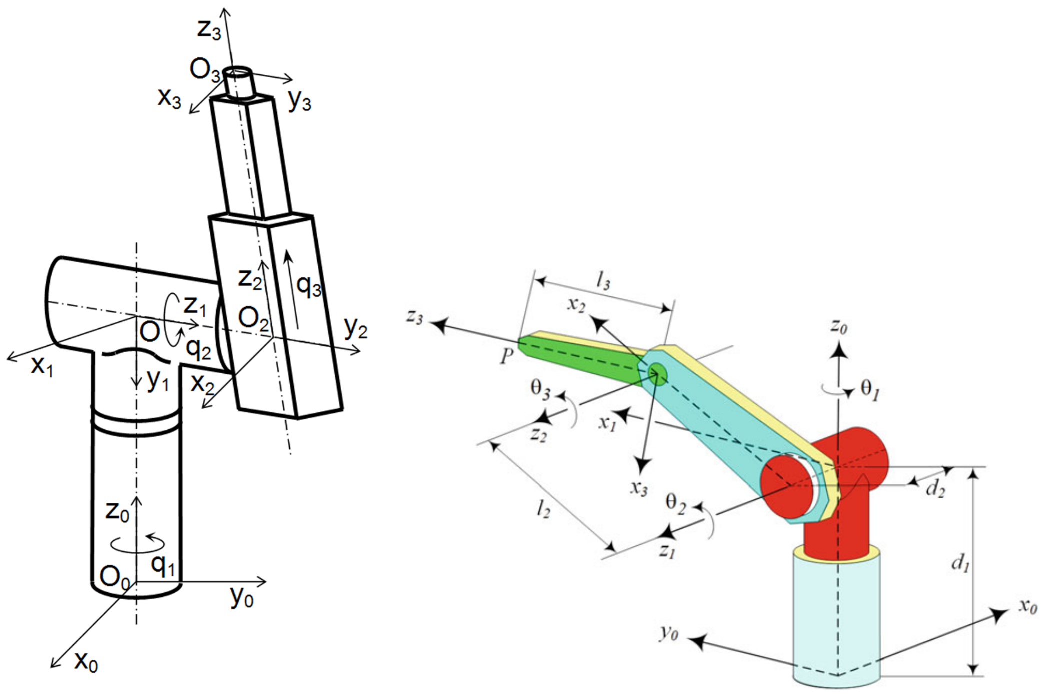
Cobots [24] Collaborative Robots, commonly referred to as cobots, are engineered with the specific purpose of working in tandem with, or directly alongside, human operators. Unlike many other categories of robots that function autonomously or within strictly segregated workspaces, cobots share work environments with human personnel to enhance their collective productivity. Their primary role often involves the removal of manual, hazardous, or physically demanding tasks from daily operations. In certain scenarios, cobots are capable of responding to and learning from human movements, further enhancing their adaptability.

The initial four robots fall under the category of mobile robots, possessing the capability to navigate within their surroundings, while the latter two are categorized as stationary robots, as detailed in table 2.1 below.

TABLE 2.1: Robots Classification.

Mobile	**Stationary**
AMRs AGVs Humanoids Hybrids	Articulated robots **Cobots**

Within the scope of this paper, our exclusive focus will be on **cobots** [24]. Across all the experiments conducted in this study, a cobot (Ned2, detailed and described in chapter 6) has been consistently utilized.



(a) 2R-1P manipulator. [43] (b) 3R manipulator. [13]

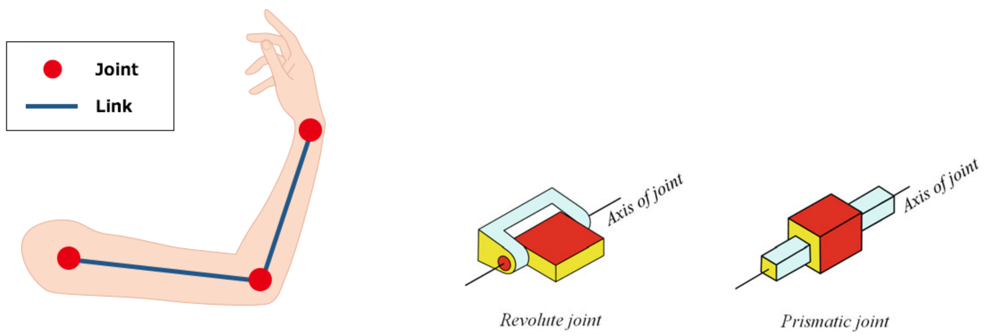
FIGURE 2.2: Manipulator with 3 DoF of movement.

2.2 Robot Components

In our study, we establish a kinematic model for a robotic manipulator, which is essentially a multi-body system comprising interconnected rigid bodies. These bodies are connected through revolute or prismatic joints (only revolute in our application), enabling relative movement. We employ principles of rigid body kinematics to elucidate the relative motions between these interconnected bodies.

It's imperative to note that a comprehensive robotic system encompasses not only the manipulator or rover but also components such as the wrist, end-effector, actuators, sensors, controllers, processors, and software. [13]

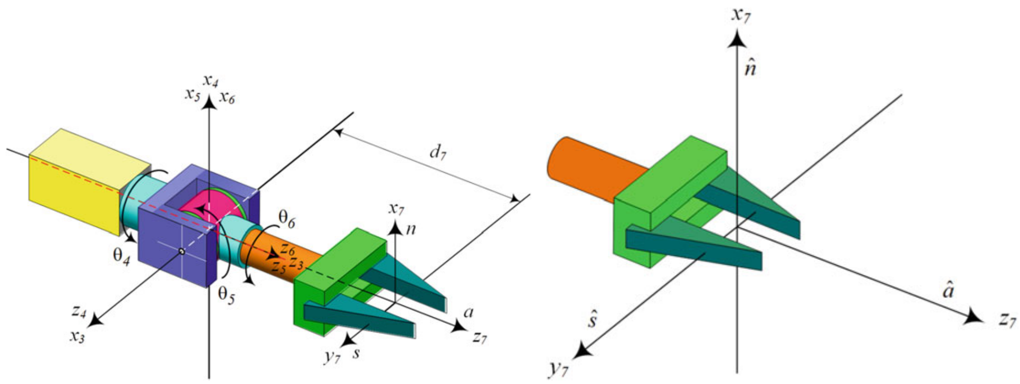
Link: In the realm of robotics, each individual rigid component within a robot that



(a) A human elbow and shoulder are joints and the bones connecting them are links. [16]

(b) Revolute and Prismatic joints.

FIGURE 2.3: Illustration of different joints.



(a) A spherical wrist kinematics. (b) End-effector.

FIGURE 2.4: The Wrist and End-effector. [13]

possesses the capacity to move concerning all other components is formally known as a 'link.' This terminology accommodates various descriptions, including 'bar,' 'arm,' or any object deemed equivalent to a link in the context of robot mechanics. A robot arm or link, in essence, represents a solid, rigid element capable of relative motion when compared to the other links within the robotic structure.

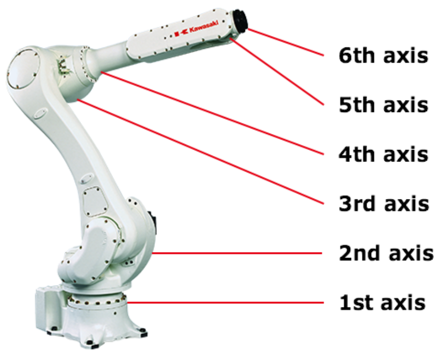
Moreover, when we encounter two or more linked components that are entirely constrained in terms of relative movement, they are collectively regarded as a 'compound link,' forming a unified and motionally inseparable entity within the robot's framework.

Joint: A kinematic pair refers to two rigid bodies that are in constant contact with one another, enabling potential relative motion. These pairs connect two links at a joint where their relative movement is described by a single joint coordinate. Typically, joints manifest as revolute (rotary) or prismatic (translatory). In Figure 2.3(a), the illustration portrays joints and links within a human arm. A revolute joint (R) mimics a hinge, enabling relative rotation between two links, while a prismatic joint (P) facilitates translational relative motion between the two links.

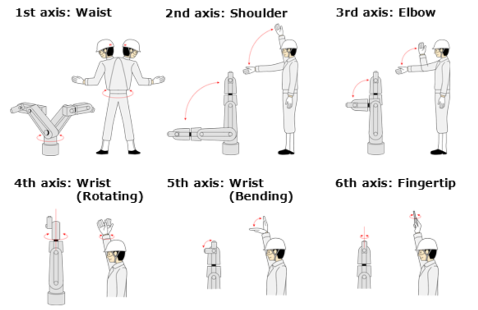
The relative rotation of interconnected links through a revolute joint transpires along an axis referred to as the joint axis. Similarly, the translation between two connected links via a prismatic joint occurs along an axis, also known as the joint axis. The joint coordinate, or joint variable, represents the value describing the relative position of these connected links at the joint. For a revolute joint, it represents an angle, while for a prismatic joint, it denotes a distance. The depiction of revolute and prismatic joints is respectively shown in Figures 2.3(b).

Manipulator: The primary structure of a robot comprises links, joints, and associated structural components, known as the manipulator. A manipulator is transformed into a robot when integrated with a wrist, gripper, and its control system. However, in the literature, robots and manipulators are often used interchangeably, both denoting robotic systems. In Figure 2.2(b), a 3R manipulator is depicted, while Figure 2.2(a) showcases a 2R-1P manipulator arm.

Wrist: The joints situated in the kinematic chain between the forearm and the end-effector are termed the wrist. It's customary to design manipulators with



(a) Kawasaki Cobot.



(b) The respective motion of each axis compared to human motion.

FIGURE 2.5: How a Cobot range of motion and DoF are inspired by human motion. [16]

spherical wrists, comprising three intersecting revolute joint axes that converge at a common point known as the wrist point. Illustrated in Figure 2.4(a) is a stationary spherical wrist composed of three revolute joints with orthogonal rotation axes. The spherical wrist significantly streamlines kinematic analysis by separating the end-effector's positioning and orientation. Consequently, the manipulator's wrist point holds three degrees of freedom for positioning, controlled by three arm joints. The orientation's degrees of freedom depend on the wrist design, which may involve one, two, or three degrees based on the specific application.

End-Effector: The end-effector constitutes the segment attached to the final link that carries out the designated tasks of the robot. Typically, the simplest end-effector is a gripper, allowing two actions: opening and closing. Both the arm and wrist assemblies primarily facilitate the positioning of the end-effector and any accompanying tools. It is the end-effector that executes the intended tasks. Considerable research is dedicated to developing specialized end-effectors and tools, including anthropomorphic hands designed for prosthetic purposes in manufacturing. Consequently, a robot is comprised of a manipulator or mainframe, a wrist, and an end-effector, while the wrist and end-effector assembly is sometimes referred to as a hand. Figure 2.4 presents an example of an end-effector.

Actuator: Actuators serve as the driving force akin to a robot's muscles, enabling the alteration of their configuration. They supply the necessary power to act on the mechanical structure, counteracting external forces like gravity and inertia, to adjust the geometric position and orientation of the robot's hand. Actuators, typically electric, hydraulic, or pneumatic, must be controllable.

Sensors: Sensors are components employed to observe and gather data related to both internal and external conditions. Within the context of this book, pivotal data such as joint positions, velocities, accelerations, and forces hold significant importance for measurement. Embedded within the robot system, these sensors relay pertinent details concerning each link and joint to the control unit. The control unit, based on this information, ascertains the robot's configuration.

Controller: The robot's controller, or control unit, fulfills three primary functions:

1. Information role, involved in the collection and processing of data obtained from the robot's sensors.
2. Decision role, responsible for planning the geometric movement of the robot structure.
3. Communication role, tasked with managing information exchange between the robot and its environment. The control unit encompasses both the processor and software components.

2.3 Robot Kinematics

2.4 Robot Control (Software)

Chapter 3

Robot Operating System | ROS

Definition

ROS serves as an open-source, meta-operating system designed to cater to the requirements of your robot. Much like traditional operating systems, it offers essential functionalities such as hardware abstraction, precise control over low-level devices, integration of commonly-used features, seamless message exchange between processes, and streamlined package management. Furthermore, ROS equips you with a comprehensive set of tools and libraries to facilitate tasks like code acquisition, compilation, development, and execution across diverse computing environments. [28]

In this chapter, we will delve into the foundational elements of the ROS framework. We will begin by exploring the relevance and basics of Linux and Ubuntu in the context of ROS. Following that, we will delve into the philosophical underpinnings of ROS, its master communication structure, and how these elements relate to topics, services, and actions. Finally, we will provide an explanation of the motion planning framework known as MoveIt.

3.1 Linux for Robotics

Linux is a free, open-source operating system that includes several utilities that will significantly simplify your life as a robot programmer. As will be shown in the upcoming sections, ROS (Robot Operating System) is based on a Linux system. All commands and concepts explained here are taken from the Linux tutorial made by the University of Surrey. [14]

3.1.1 What Is Ubuntu? and Why for Robotics?

Ubuntu, accessible at `www.ubuntu.com`, stands as a widely acclaimed Linux distribution rooted in the Debian architecture (source: `https://en.wikipedia.org/wiki/Debian`). Notably, it's freely available and open source, permitting extensive customization for specific applications. Ubuntu boasts an extensive software repository, comprising over 1,000 software components, encompassing essentials such as the Linux kernel, GNOME/KDE desktop environments, and a suite of standard desktop applications, including word processing tools, web browsers, spreadsheets, web servers, programming languages, integrated development environments (IDEs), and even PC games. Versatile in its deployment, Ubuntu can operate on both desktop and server platforms, accommodating architectures like Intel x86, AMD-64, ARMv7,

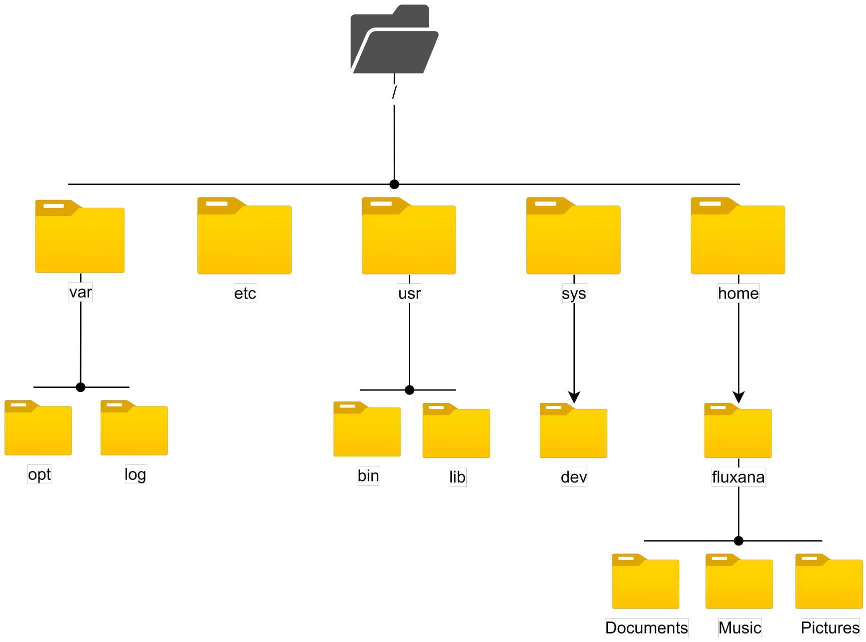


FIGURE 3.1: Ubuntu file system structure

and ARMv8 (ARM64). Canonical Ltd., headquartered in the UK (www.canonical.com), provides substantial backing to Ubuntu. [18]

In the realm of robotics, software stands as the nucleus of any robotic system. An operating system serves as the foundation, facilitating seamless interaction with robot actuators and sensors. A Linux-based operating system, such as Ubuntu, offers unparalleled flexibility in interfacing with low-level hardware while affording provisions for tailored OS configurations tailored to specific robot applications. Ubuntu's merits in this context are manifold: it exhibits responsiveness, maintains a lightweight profile, and upholds stringent security measures. Additionally, Ubuntu boasts a robust community support ecosystem and a cadence of frequent releases, ensuring its perpetual relevance. It also offers long-term support (LTS) releases, guaranteeing user assistance for up to five years. These compelling attributes have cemented Ubuntu as the preferred choice among developers in the Robot Operating System (ROS) community. Indeed, Ubuntu stands as the sole operating system that enjoys comprehensive support from ROS developers. The Ubuntu-ROS synergy emerges as the quintessential choice for programming robots. [14]

3.1.2 Ubuntu File Structure

Similar to the 'C drive' in a Windows operating system, Linux incorporates a dedicated storage area for its system files, known as the root file system. This root file system is established during the Ubuntu installation process, with the assignment of '/' as its designated mount point. For a visual representation of the Ubuntu file system architecture, refer to Figure 3.1.

The following describes the uses of each folder in the file system:

- `/bin and /sbin`: These directories house essential system applications, akin to the 'C:/Windows' folder in Windows.
- `/etc`: Within this directory, system configuration files are stored.

- */home/yourusername*: Equivalent to the 'C:/Users' directory in Windows, this directory serves as the user's home.
- */lib*: Similar to '.dll' files in Windows, the '/lib' directory contains library files.
- */media*: This directory serves as the mount point for removable media.
- */root*: The '/root' directory contains files associated with the root user, who holds administrative privileges in the Linux system.
- */usr*: Pronounced 'user,' the '/usr' directory hosts a majority of program files, akin to 'C:/Program Files' in Microsoft Windows.
- */var/log*: Within this directory, you'll find log files generated by various applications.
- */home/yourusername/Desktop*: The location for Ubuntu desktop files.
- */mnt*: Mounted partitions are accessible in this directory.
- */boot*: This directory stores essential files required for the boot process.
- */dev*: Linux device files are located here.
- */opt*: The '/opt' directory serves as the designated location for optionally installed programs. (For instance, ROS is installed in '/opt').
- */sys*: This directory houses files containing critical information about the system.

3.2 Philosophy Behind ROS

The philosophical objectives of ROS can be succinctly described as follows [26]:

- Decentralized collaboration: Emphasizing **peer-to-peer** interactions. ROS consists of multiple interconnected software components enabling continuous message exchange without centralized routing, allowing for scalability even with increased data volume, although it may add complexity to the system infrastructure.
- **Tool-oriented** approach: Focusing on the development of a robust set of tools. ROS, influenced by Unix's architectural principles, constructs complex software systems through multiple small, versatile programs, differentiating itself from other robotics software by decentralizing various tasks to discrete software, enabling continual evolution and tool enhancement for specific task domains.
- **Multilingual support**: Enabling compatibility with multiple programming languages. ROS embraces a multilingual approach, allowing modules to be written in diverse programming languages, including C++, Python, LISP, Java, JavaScript, MATLAB [6], Ruby, Haskell, R, Julia, and others, with this book predominantly using the Python client library for code examples, yet noting the flexibility to utilize other available client libraries for discussed tasks.

- **Thin** design: Prioritizing a streamlined framework. ROS conventions encourage an approach where developers build standalone libraries that can be incorporated into ROS modules, fostering versatile software reuse and enabling streamlined automated testing via established continuous integration tools.

- Openness and freedom: Being freely available and based on **open-source** principles. ROS, operating under the permissive BSD license [27], enables diverse licensing arrangements by allowing flexible integration of closed-source and open-source modules, accommodating commercial, academic, and hobby projects, ensuring compliance within varied environments [26].

To the best of our knowledge, no existing framework encompasses this specific set of design principles. This section aims to delve into these philosophies, elucidating how they have profoundly influenced the design and implementation of ROS [26].

3.3 Preliminaries

Before delving into ROS, it's essential to introduce the fundamental concepts that underpin this framework. ROS systems are constituted by a multitude of autonomous programs that maintain continuous communication with one another. This section provides an in-depth exploration of this architectural setup and the associated command-line tools. It further delves into the intricate aspects of ROS naming conventions and namespaces, demonstrating their role in facilitating code reusability. [25]

3.3.1 ROS-Graph [31]

One of the original challenges inspiring the creation of ROS was commonly known as the 'fetch an item' problem. This scenario involved a relatively large and complex robot equipped with various sensors, a manipulator arm, and a mobile base. In the 'fetch an item' problem, the robot's objective is to navigate a typical home or office environment, locate a specified item, and transport it to the designated location. This task led to several key observations, which subsequently became foundational design goals for ROS:

- The application task can be broken down into numerous autonomous subsystems, encompassing areas like navigation, computer vision, and grasping.

- These subsystems are adaptable for various tasks, such as security patrols, cleaning, and mail delivery, among others.

- By implementing appropriate hardware and geometry abstraction layers, the majority of application software can be made compatible with different robotic platforms.

These principles are exemplified through the core structure of a ROS system: its graphical representation. In ROS, multiple programs operate concurrently and communicate by exchanging messages. This system structure is conveniently portrayed as a mathematical graph, with nodes representing individual programs and edges indicating their communication. While Figure 3.2 illustrates a sample ROS graph from one of the early 'fetch an item' implementations, the specific details are less significant compared to the overarching concept of a ROS system as an assembly of

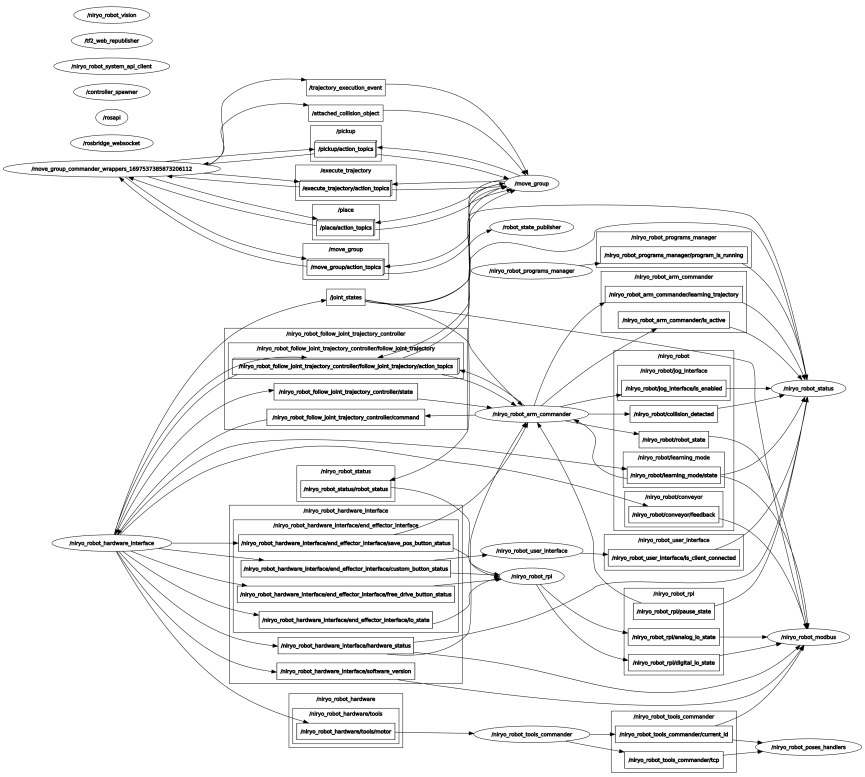


FIGURE 3.2: Graphical representation of a ROS system for 'Niryo-Ned2 [9]' robot—nodes/topics within the graph symbolize individual software modules, while edges denote message streams facilitating the exchange of sensor data, actuator commands, planner states, intermediate representations, and other relevant information.

nodes engaged in message-based communication. This representation serves as a practical framework for software development, emphasizing the modular nature of ROS programs, or 'nodes,' as integral components within a larger system.

In summary, within a ROS graph, a node signifies a software module engaged in message transmission, and an edge denotes the flow of messages between two nodes. While complexity can increase, nodes are typically POSIX processes, and edges are akin to TCP connections, enhancing fault tolerance as a software crash typically affects only the crashing process, leaving the rest of the graph operational. The events leading to the crash can often be reconstructed by logging messages entering a node and replaying them within a debugger at a later time.

One of the most significant advantages of a loosely coupled, graph-based architecture is the capacity to rapidly prototype complex systems with minimal or no need for additional 'glue' software during experimentation. Individual nodes, such as the object recognition node in a 'fetch an item' system, can be effortlessly replaced by launching an entirely different process that handles images and generates labeled objects. Beyond node replacement, entire segments of the graph (subgraphs) can be dynamically dismantled and substituted with other subgraphs in real time. This flexibility extends to replacing real-robot hardware drivers with simulators, swapping navigation subsystems, fine-tuning algorithms, and more. Since ROS dynamically generates the necessary network backends, the entire system fosters an interactive environment that encourages experimentation.

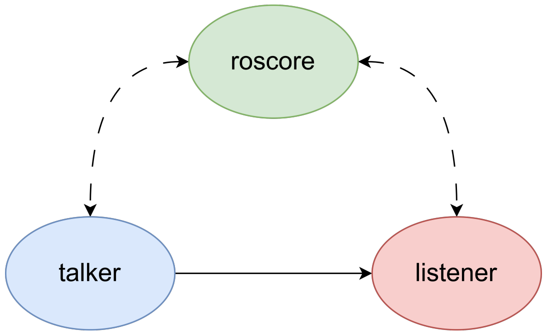


FIGURE 3.3: roscore establishes ephemeral connections with the other nodes in the system.

To this point, we have assumed that nodes discover each other, but we have not elaborated on the process. Amidst the extensive network traffic, how do nodes locate and initiate message exchange? The solution lies in a program known as 'roscore'.

3.3.2 Roscore [30]

roscore serves as a vital component within the ROS ecosystem by facilitating connections between nodes to enable message transmission. During initialization, each node registers its published message streams and desired subscriptions with roscore, allowing it to establish direct peer-to-peer connections with other nodes participating in the same message topics. A functioning roscore is imperative for any ROS system since it serves as a vital reference point for nodes to discover one another.

It's important to note that while roscore plays a crucial role in aiding nodes in locating their peers, the actual message transmission between nodes occurs in a peer-to-peer manner. This setup can sometimes be misconstrued, especially for individuals accustomed to client/server systems from web-based backgrounds, wherein the roles of clients and servers are more distinct. The ROS architecture, however, functions as a hybrid system, integrating aspects of both client/server and fully distributed models, thanks to the central role of roscore, which acts as a naming service for peer-to-peer message streams.

When a ROS node initiates, it relies on the presence of an environment variable, ROS_MASTER_URI, which should contain a URL of the form http://hostname:11311/. This URL signifies the existence of a functioning roscore accessible on port 11311, hosted on a machine named hostname, which can be reached over the network.

With this information, nodes communicate with roscore at startup to register themselves and query for other nodes and message streams by name. Each node informs roscore about the messages it can provide and those it wishes to subscribe to. roscore, in return, supplies the necessary details about the message producers and consumers. In graphical terms, every node within the system can periodically utilize roscore services to identify and connect with its peers. This is illustrated by the dashed lines in Figure 3.3, signifying that in a basic two-node setup, the talker and listener nodes intermittently make service calls to roscore while directly engaging in peer-to-peer message exchange.

Roscore also serves as a parameter server, extensively utilized by ROS nodes for configuration purposes. It enables nodes to store and retrieve various data structures, including robot descriptions and algorithm parameters. To interact with the parameter server, ROS provides a command-line tool called 'rosparam,' which we will use throughout this book.

We will delve into examples of using roscore shortly. For now, it's essential to remember that roscore facilitates nodes in discovering other nodes. Before we proceed to run some nodes, it's worth understanding how ROS organizes packages and gaining some insight into the ROS build system, known as 'catkin. [25]

3.3.3 catkin, Workspaces, and ROS Packages

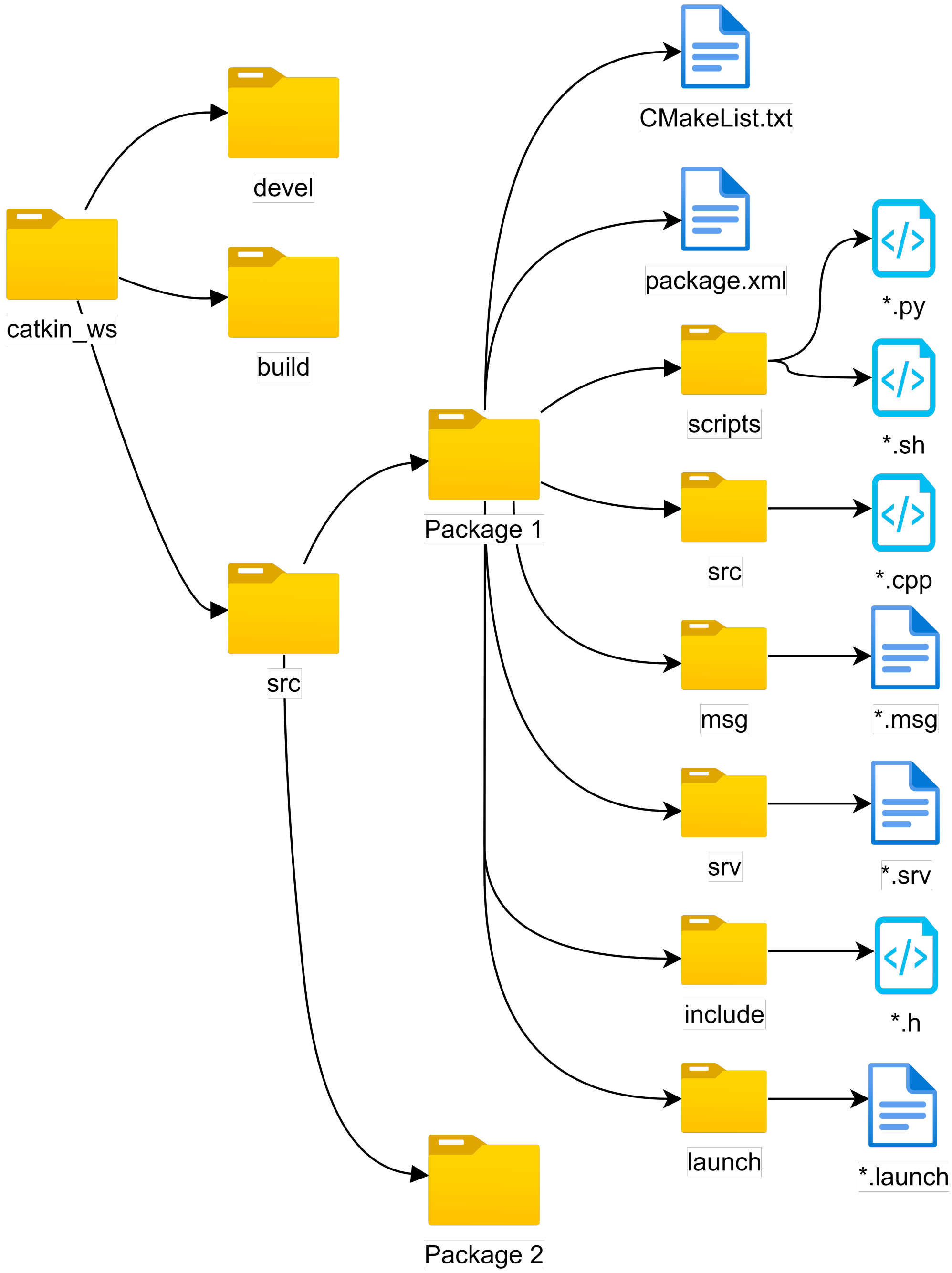


FIGURE 3.4: Files' structure for ROS workspace. [2]

`Catkin` serves as the ROS build system, comprising a set of tools utilized by ROS for generating executable programs, libraries, scripts, and interfaces that can be employed by other code. If you are developing your ROS code in C++, a good understanding of catkin is essential. However, since this book employs Python for its examples, we won't delve deeply into its intricacies. Nevertheless, we will explore its basic functionalities to some extent. For those interested in a more comprehensive understanding, the catkin wiki page is an excellent resource. If you are curious about why ROS has its dedicated build system, you can refer to the catkin conceptual overview wiki page. To install ros-melodic catkin workspace, you can use the following command:

Command

```
sudo apt-get install ros-melodic-catkin
```

ROS Catkin Workspace

The ros workspace has several folders as shown in Figure 3.4 above. Following, we will be looking at the function of each folder. [15]

src Forlder The 'src' directory within the catkin workspace serves as the designated location for creating or importing new packages from repositories. It's important to note that ROS packages are only built and turned into executables when they reside in the 'src' directory. When the 'catkin_make' command is executed from the workspace directory, it scans the 'src' folder, building each package found there.

build Folder When the 'catkin_make' command is executed within the ROS workspace, the catkin tool generates certain build files and intermediate CMake cache files within the 'build' directory. These cache files play a crucial role in preventing the need to rebuild all packages each time you run 'catkin_make.' For example, if you initially build five packages and subsequently introduce a new package to the 'src' folder, only the new package will be built during the next 'catkin_make' command. This efficiency is achieved through the utilization of cache files within the 'build' directory. It's important to note that deleting the 'build' folder will trigger a complete rebuild of all packages.

devel Folder When 'catkin_make' is executed, it triggers the build process for each package, resulting in the creation of target executables if the build is successful. These executables are saved within the 'devel' folder, which contains shell script files designed to incorporate the current workspace into the ROS workspace path. Access to the packages within the current workspace is only enabled when this script is executed. Typically, the following command is employed for this purpose.

Command

```
source ~/<workspace_name>/devel/setup.bash
```

3.4 ROS Master Communication

The ROS Master functions as a provider of naming and registration services for the various nodes within the ROS system. It manages the tracking of publishers and subscribers to topics, as well as services. Essentially, the role of the Master is to facilitate the discovery of individual ROS nodes, allowing them to establish peer-to-peer communication.

Additionally, the Master offers the Parameter Server functionality.

To initiate the Master, the 'roscore' command is commonly used, which initiates the ROS Master alongside other indispensable components.

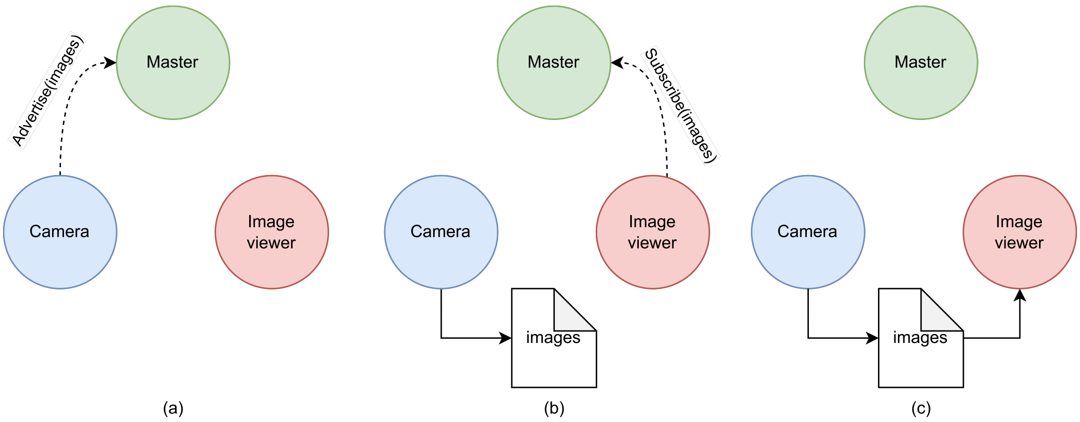


FIGURE 3.5: Master Name Service Example [29]

In the above-shown figure 3.5, let's consider a scenario involving two Nodes: a Camera node and an Image_viewer node. The typical sequence of events begins with the Camera node informing the master that it intends to publish images on the 'images' topic, as depicted in (a).

Subsequently, the Camera node commences publishing images to the 'images' topic. However, as there are no subscribers to this topic, no data is transmitted. In (b), the Image_viewer expresses its interest in subscribing to the 'images' topic to check if any images are available.

With both a publisher and a subscriber now present for the 'images' topic, the master node facilitates the awareness of Camera and Image_viewer to each other's existence, allowing them to commence the exchange of images, as illustrated in part (c) of the figure.

3.4.1 .bashrc File

It's essential to highlight the significance of the .bashrc file for comprehending the ROS-master communication setup. As discussed in the roscore section (3.3.2), several environment variables, such as ROS_MASTER_URI, are established during roscore initialization, often containing a URL in the format `http://hostname:11311/`, where the `hostname` represents the IP address of the ROS-master computer. The .bashrc, an abbreviation for **bash r**ead **c**ommand, serves as a configuration file for the Bash shell environment. Upon the initiation of an interactive Bash shell session, the .bashrc script file executes, encompassing diverse comments, configurations, **environment variable settings**, and functions aimed at customizing the shell experience and automating tasks.

3.4.2 Publishers-Subscribers

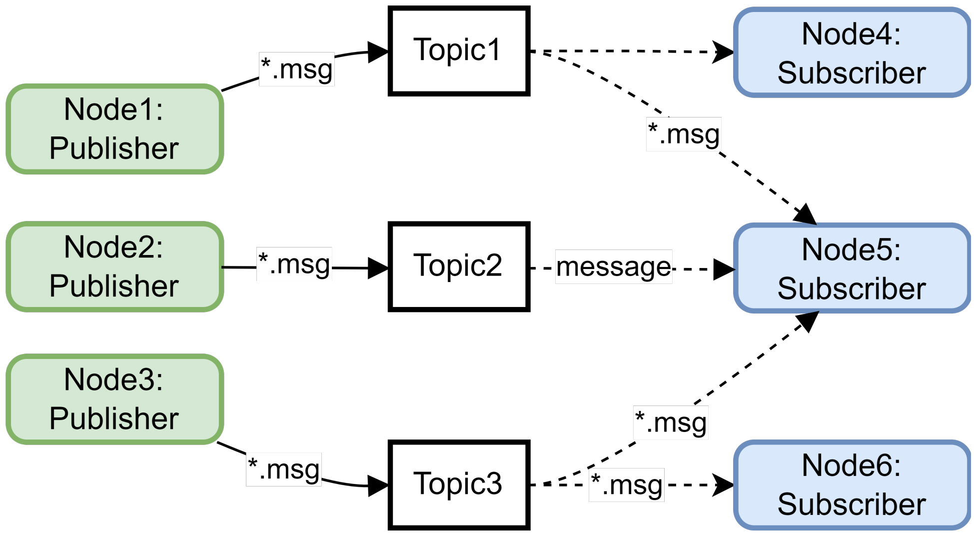


FIGURE 3.6: Publish–subscribe communication model.

3.4.3 Services

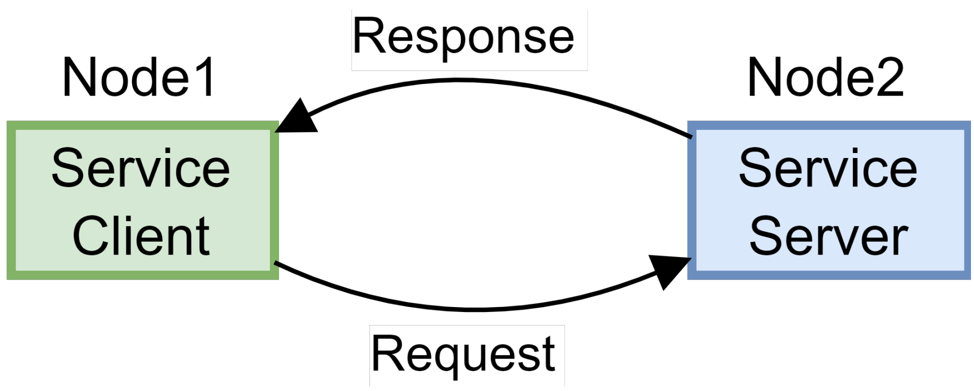


FIGURE 3.7: Server-Client Service communication

3.4.4 Actions

3.5 MoveIt! [36]

MoveIt![5] serves as the primary software framework within the Robot Operating System (ROS) for motion planning and mobile manipulation. It has garnered acclaim for its seamless integration with various robotic platforms, including the PR2 [45], Robonaut [1], and DARPA's Atlas robot. MoveIt! is

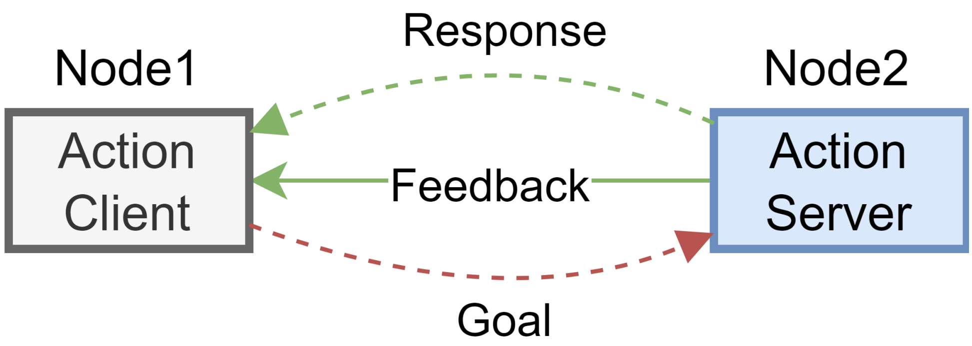


FIGURE 3.8: Server-Client Action communication

primarily coded in C++, augmented by Python bindings to facilitate higher-level scripting. Embracing the fundamental principle of software reuse, advocated for in the realm of robotics [21], MoveIt! adopts an agnostic approach towards robotic frameworks, such as ROS. This approach entails a formal separation between its core functionality and framework-specific elements, ensuring flexibility and adaptability, especially in inter-component communication.

By default, MoveIt! leverages the core ROS build and messaging systems. To facilitate effortless component swapping, MoveIt! extensively employs plugins across its functionality spectrum. This includes motion planning plugins (currently utilizing OMPL), collision detection (presently incorporating the Fast Collision Library (FCL) [23]), and kinematics plugins (employing the OROCOS Kinematics and Dynamics Library (KDL) [33] for both forward and inverse kinematics, accommodating generic arms alongside custom plugins).

MoveIt!'s principal application domain lies in manipulation, encompassing both stationary and mobile scenarios, across industrial, commercial, and research settings. For a more comprehensive exploration of MoveIt!, interested readers are encouraged to refer to [5].

Chapter 4

Basics of Powder Dosing

4.1 Affecting Parameters

Part II

Experimental Set-up, Methodology, and Results

Chapter 5

Methodology

In this chapter, we will elucidate the methodology employed in conducting the powder dosage experiments. It will not only encompass crucial Python code with MoveIt motion planning algorithms but also outline the comprehensive sequence logic governing the entire experiment. Additionally, we'll clarify significant functions utilized in constructing the sequence logic and algorithm before delving into the algorithm itself to ensure better comprehension.

5.1 Sequence Logic

In industrial automation, sequence logic refers to a meticulously arranged series of actions orchestrated to accomplish a particular outcome in a methodical and organized manner. These sequences serve as structured procedures executed by automated systems to achieve defined objectives. [7]

The sequential logic can be outlined in bullet points as follows:

1. **Start**: Calibration involves the automatic definition of the Zero positions for each axis of the robot.
2. Retrieve the Pre-saved trajectories.
3. Balance Setup.
4. Obtain the glass crucible.
5. Configure the dosing parameters.
6. Control the powder dosing process while receiving feedback from the balance.
7. Return the crucible upon completion of the dosing process.
8. Unset the balance, repositioning the metal crucible.
9. **End**: Return to the home position.

Each of these sequences will be explained in detail in the state diagram section 5.3.

5.2 Important Fuctions from FX_ROS Package

The FX_ROS.py package, developed by me, is dedicated to performing all the computational tasks related to motion planning using MoveIt and handling various ROS communication operations. This package efficiently manages operations such as subscribing to specific topics and initializing a ROS service client to invoke service servers for designated actions. It centralizes and streamlines these diverse functionalities within a single framework for enhanced convenience and effective ROS system interaction.

Throughout this section, we will dive into the most important functions in order to fully understand the state diagram of the sequence logic in the next section.
For more extensive details about the full script of FX_ROS package, please refer to Appendix B.

5.2.1 Calling a Service Server

As shown in the code below, rather than

```python
def Call_Aservice(service_name, type, request_name=None,req_args=None):
    """Call a ROS service.

    Parameters:
    ........................
    service_name: str
    type: srv
    request_name: None (srv)
    req_args: None (dictionary) ex. {'positon': 210, 'id': 11, 'value': False}

    Returns:
    ........................
    Return the response of the service.
    """
    try:
        rospy.wait_for_service(service_name, 2)
    except (rospy.ServiceException, rospy.ROSException) as e:
        rospy.logerr("Timeout and the Service was not available : " + str(e))
        return RobotState()

    try:
        service_call = rospy.ServiceProxy(service_name, type)
        if request_name == None:
            response = service_call()
        else:
            request = request_name()
            for key, value in req_args.items():
                #print("f{key} = {value}")
                method = setattr(request, key, value)
            response = service_call(request)
    except rospy.ServiceException as e:
        rospy.logerr("Falied to call the Service: " + str(e))
        return 0

    return response
```

LISTING 5.1: Call_Aservice Function

Example of Call_Aservice Function

The code snippet presented below serves as an illustration of the ease with which the Call_Aservice function is employed. This example specifically functions as a service client aimed at calibrating the motors of the Ned2 robot, a crucial step preceding any motion planning. Refer to section 5.3.1 for comprehensive insights into the motor calibration process.

```
def motor_cal():
    cal_service = '/niryo_robot/joints_interface/calibrate_motors'
    Call_Aservice(cal_service, type=SetInt, request_name=SetIntRequest,
     req_args={"value":1})
```

LISTING 5.2: motor_cal Function

5.2.2 Subscribing to Topic

```
def Subscribe(topic_name, type, msg_args):
    """Subscribe to a certain topic.

    Parameters:
    ........................
    topic_name: str
    type: srv
    msg_args: list >> list of strings, which contains the arguments
   that we need to read from the topic.

    Returns:
    ........................
    Return a list of the read values from each argument.
    If we have only one argument, it returns the value of this argument
     only, not a list.
    """
    try:
        msg = rospy.wait_for_message(topic_name, type, 2)
    except:
        rospy.logerr("Timeout and the Topic Did not recieve any
   messages")
        return 0

    value = []
    if len(msg_args) == 1:
        value = getattr(msg, msg_args[0])
    else:
        for i in msg_args:
            value.append(getattr(msg, i))

    return value
```

LISTING 5.3: Subscribe Fucntion

Examples of Subscribe Function

The subsequent functions demonstrate the straightforward process of subscribing to a particular topic and retrieving the intended data. In the first case, the topic "/joint_states" acquires its published information concerning the present joint positions from the sensors node. Furthermore, the get_pose function is designed to subscribe to the "/nirio_robot/robot_state" topic, which disseminates a RobotState message.

```python
def Get_joints():
    """return a tuple of 6 values for each joint from 1 till 6"""

    joints_values = Subscribe('/joint_states', JointState, ["position"])
    return joints_values
```

LISTING 5.4: Get_joints Function

```python
def get_pose():
    """Gets the pose values from the robot_state topic.
    Return:
    ..................................
    a list of two dictionaries, the first is positions (x,y,z),
    whereas the second is the rpy (roll, pitch, yaw)
    """

    return Subscribe('/niryo_robot/robot_state', RobotState, ['position', 'rpy'])
```

LISTING 5.5: get_pose Function

5.2.3 Moveing the Cobot using Joints

```python
arm = moveit_commander.move_group.MoveGroupCommander("arm")
```

```python
def move_to_joints(joints, arm_speed=None):
    """Move to a given joint values.
    Parameters:
    ........................

    joints: list or tuble -> [joint1, joint2, joint3, joint4, joint5, joint6]
    arm_speed: float (optional) -> between 0 and 1. (0,1]
    """
    joints_limits = Get_Joints_limits()

    for i in range(6):
        if joints_limits.joint_limits[i].max < joints[i] or joints[i] < joints_limits.joint_limits[i].min:
            print("Joint{} = {}, which is out of limit!".format(i+1, joints[i]))
            print("Joint{} can not be more than {} neither less than {}".format(i+1, joints_limits.joint_limits[i].max, joints_limits.joint_limits[i].min))
            return
        else:
            pass

    #arm.set_joint_value_target(joints)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(joints, wait=True)

    arm.stop()
```

LISTING 5.6: Moving to a certain joints values

```python
def Move_joint_axis(axis, new=None, add=None, arm_speed=None):
    """You should either put a value to add or new, not both.

```

```
    Parameters:
    ........................
    * axis: int -> the number of the joint that you want to move

    * new: float -> The new coordination you want to give to a joint (axis).
            "new" will always overright the value of the axis.
    * add: float -> the value in meters change in a certain joint (axis).
    * arm_speed: float (optional) -> between 0 and 1. (0,1]

    Returns: None
    ........................
    """
    moving_joints = list(Get_joints())

    if new:
        moving_joints[axis-1] = new
    elif add:
        moving_joints[axis-1] += add

    joints_limits = Get_Joints_limits()

    if joints_limits.joint_limits[axis-1].max < moving_joints[axis-1] or moving_joints[axis-1] < joints_limits.joint_limits[axis-1].min:
        print("The joint{} can not be more than {} neither less than {}".format(axis, joints_limits.joint_limits[axis-1].max, joints_limits.joint_limits[axis-1].min))
        return 0
    else:
        pass

    arm.set_joint_value_target(moving_joints)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(moving_joints, wait=True)

    arm.stop()
```

LISTING 5.7: Moving only one axis in joints

5.2.4 Moveing the Cobot using Pose

```
def Move_to_pose(pose_values, arm_speed=None):
    """Move to a given pose values.
    Parameters:
    ........................

    pose_values: list or tuble -> [x, y, z, roll, pitch, yaw]
    arm_speed: float (optional) -> between 0 and 1. (0,1]
    """

    pose = Pose()
    p_goal = pose.position
    orn_goal = pose.orientation

    p_goal.x = pose_values[0]
    p_goal.y = pose_values[1]
    p_goal.z = pose_values[2]

    roll = pose_values[3]
```

```
    pitch = pose_values[4]
    yaw   = pose_values[5]

    orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
    .quaternion_from_euler(roll,pitch,yaw)

    #arm.set_goal_tolerance(0.001)
    if arm_speed:
        set_speed(arm_speed)
    arm.set_pose_target(pose)
    arm.go(wait=True)

    arm.stop()
    arm.clear_pose_targets()
```

LISTING 5.8: Moving to certain pose values of the TCP

```
def Move_pose_axis(axis, new=None, add=None, arm_speed=None):
    """You should either put a value to add or new, not both.

    Parameters:
    ........................
    * axis: str -> (x, y, z, roll, pitch, or yaw)
    * new: float -> The new coordination you want to give to a certain
    axis.
              "new" will always overwrite the value of the axis.
    * add: float -> the value in meters or radians you want to add to a
     certain axis.
    * arm_speed: float (optional) -> between 0 and 1. (0,1]
    Returns: None
    ........................
    """
    FK = get_pose()
    axises = ['x','y','z']

    pose = Pose()
    p_goal = pose.position
    orn_goal = pose.orientation

    p_current = FK[0]

    rpy_current = FK[1]

    if add:
        if axis.lower() in axises:
            current_value = getattr(p_current, axis)
            setattr(p_current, axis, current_value+add)
        else:
            current_value = getattr(rpy_current, axis)
            setattr(rpy_current, axis, current_value+add)
    if new:
        if axis.lower() in axises:
            setattr(p_current, axis, new)
        else:
            setattr(rpy_current, axis, new)


    p_goal.x = p_current.x
    p_goal.y = p_current.y
    p_goal.z = p_current.z

```

```
    orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations.quaternion_from_euler(rpy_current.roll,rpy_current.pitch, rpy_current.yaw)

    arm.set_pose_target(pose)

    if arm_speed:
        set_speed(arm_speed)
    arm.go(wait=True)

    arm.stop()
    arm.clear_pose_targets()
```

LISTING 5.9: Moving only one axis in Pose

5.2.5 Use Forward Kinematics

```
def FK_Moveit(joints):
    """Get Forward Kinematics from the MoveIt service directly after giving joints
    :param  joints
    :type   joints: list of joints values
    :return: A Pose state object
    @example of a return

position:
  x: 0.278076372862
  y: 0.101870353599
  z: 0.425462888681
orientation:
  x: 0.0257527874589
  y: 0.0122083384395
  z: 0.175399274203
  w: 0.984084775322

    """
    rospy.wait_for_service('compute_fk', 2)
    moveit_fk = rospy.ServiceProxy('compute_fk', GetPositionFK)

    fk_link = ['base_link', 'tool_link']
    header = Header(0, rospy.Time.now(), "world")
    rs = RobotStateMoveIt()

    rs.joint_state.name = ['joint_1', 'joint_2', 'joint_3', 'joint_4', 'joint_5', 'joint_6']
    rs.joint_state.position = joints

    reponse = moveit_fk(header, fk_link, rs)

    return reponse.pose_stamped[1].pose
```

LISTING 5.10: Using forward kinematics to get the pose coordination from MoveIt

5.3 State Diagram

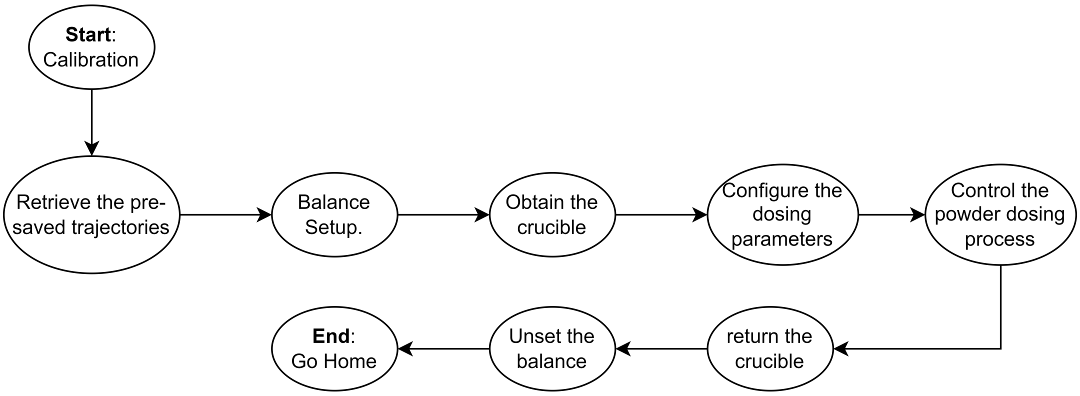


FIGURE 5.1: The sequence logic describing the full dosing process

The state diagram, resembling a state machine, comprises states, transitions, events, and activities. While the activity diagram elucidates the control flow between various activities across the involved objects, the state diagram demonstrates the control flow from one state to another within a singular object. It defines the progression of states an object experiences in reaction to events, including the corresponding responses to those events within the context of the sequence logic. [22]
In this section, we are going to understand more about each state in our sequence logic.

5.3.1 Calibration

Ned2 undergoes an automatic calibration process, essential for its control, aligning the position of each motor with its respective controller [9]. The calibration sequence involves the following steps for the initial three axes:

1. The axis rotates in the direction depicted in the accompanying image until it halts.
2. It halts upon detecting the limit.
3. The axis retraces its motion in the opposite direction.
4. The axis rotates towards the limit stop.
5. It halts upon detecting the limit stop, establishing the Zero position at this point.

FIGURE 5.2: Calibration process of Ned2 cobot.

5.3.2 Retrieve the Pre-Saved Trajectories

5.3.3 Balance Set-Up

5.3.4 Obtain the Crucible

5.3.5 Do the Experiments

5.3.6 Return the Crucible

5.3.7 Unset the Balance

5.3.8 Go home

Chapter 6

Experimental Set-up

This chapter initiates a comprehensive exploration of the experimental setup. It commences with an overview of the workspace frame, encompassing the array of interconnected devices. These include various crucibles, the balance device, both its hardware and software components, the Ned2-cobot with its hardware configurations and software tools, precision validation procedures and concludes with an examination of the vibration motor, which is an auxiliary component integrated with the cobot.

6.1 Frame

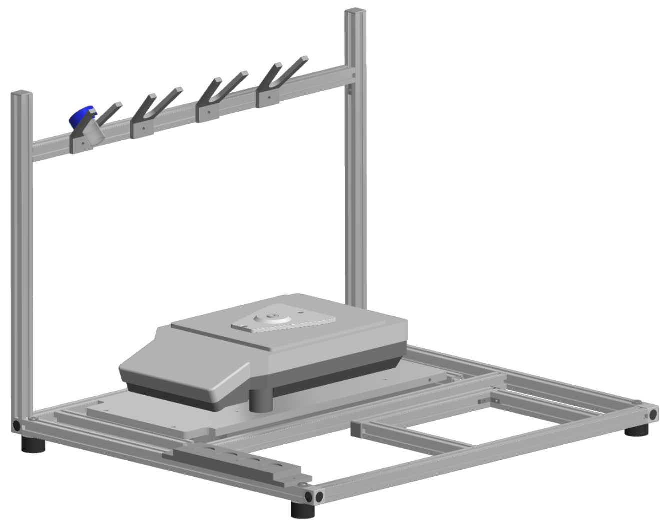
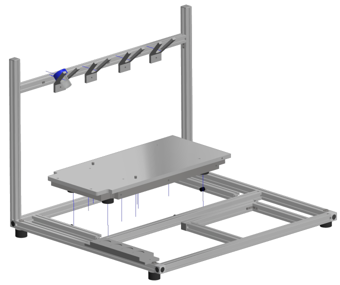


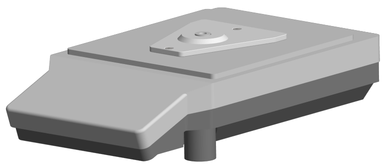
FIGURE 6.1: The Full Framework of System

In the realm of robotics, establishing a meticulously structured workspace framework is of paramount significance. This framework serves as the foundation encompassing all the requisite components with which the cobot will interact. As illustrated in Figure 6.1, this framework has been thoughtfully designed to meet these specifications.

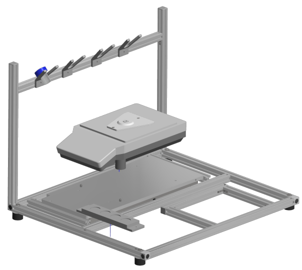
(a) The Frame



(b) Frame + ground plate for the balance



(c) The METTLER TOLEDO Balance [42]



(d) With the balance and metal crucible hanger

FIGURE 6.2: Demonstration of the parts of the framework.

The framework encompasses various crucial components, as detailed below:

1. **The Frame** itself, depicted in part (a) of Figure 6.2. More comprehensive technical documentation of the frame's constituent parts and dimensions can be found in Appendix F.

2. **The Ground Plate of the Balance**, featuring a rubber base, is illustrated in part (b) of Figure 6.2. This element plays a pivotal role in ensuring the stability of the balance. Given its high sensitivity and precision, it is imperative to eliminate any potential transmission of motion or vibration

3. The Mettler Toldedo Me Balace, showcased in part (c) of Figure 6.2. For an in-depth understanding of the balance's specifications pertinent to our project, please refer to section 6.3.

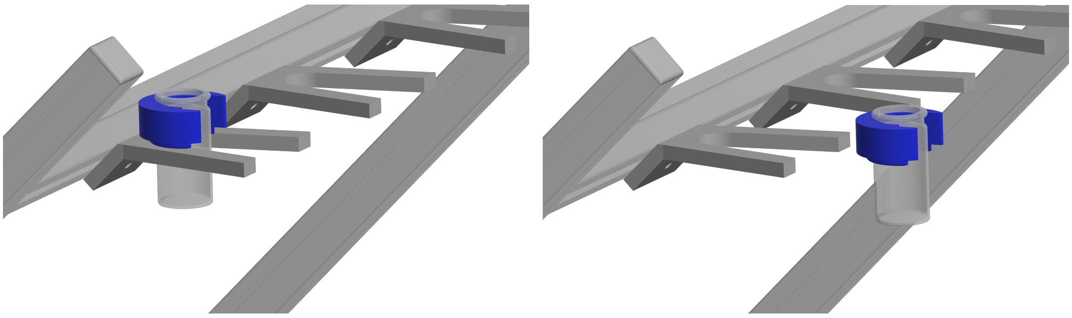
4. **Four Glass Crucibles** and their corresponding hangers, documented comprehensively in section 6.2.

5. **Four Metal Crucibles** along with their hangers, as outlined in section 6.2.

6. The **Niryo Ned2 Cobot**, expounded upon in detail in section 6.4.

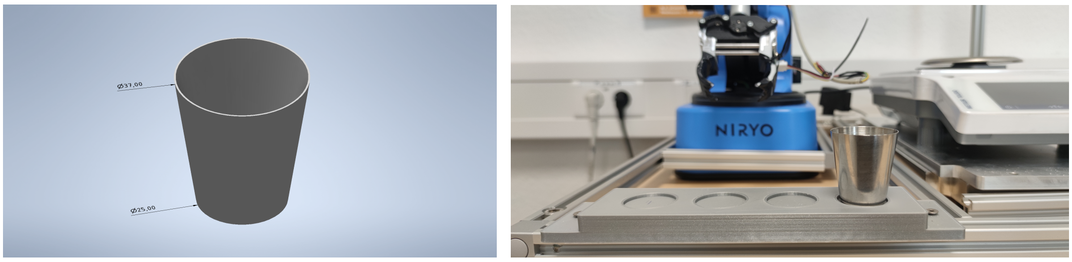
7. **Standing Rubber Buffers**, strategically incorporated to enhance the frame's stability.

Subsequent sections will delve into the intricate particulars and documentation of these essential components for our project.



(a) (b)

FIGURE 6.3: The glass crucible with its hangers.



(a) The technical design. (b) An actual crucible on the hanger.

FIGURE 6.4: The metal crucible and its hanger.

6.2 Crucibles

The crucibles play a central role in our project by serving as containers for the powder to be transferred from one crucible to another.

We will utilize two distinct types of crucibles:

- **Glass Crucibles**: These crucibles will predominantly be employed for powder dosing purposes.

  In the frame, we have incorporated four hangers for the glass crucibles, which can be observed in Figure 6.1. Additionally, the range of motion for the glass crucible is depicted in Figure 6.3. The crucible has an outer diameter of $22mm$, and an inner diameter of about $17mm$. More comprehensive information regarding the dimensions and technical specifications of the glass crucible can be found in Appendix E.

- **Metal Crucibles**: These crucibles will hold the powder after it has been dosed from the glass crucibles and will be utilized to measure the weight on the balance.

  The hanger serves the purpose of accommodating four crucibles, and you can observe its design in Figure 6.4. It possesses an upper diameter of 37mm and a lower diameter of 25mm, as visualized in Figure 6.4(a). For a more in-depth exploration of the technical design and drawings of this component, please consult the details provided in Appendix E, where comprehensive information is available.

6.3 Balance Device

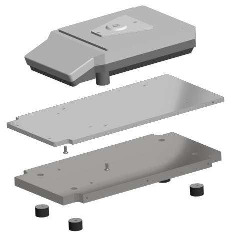


FIGURE 6.5: The balance with its metal plates and rubber buffers.

The precision balance employed in this study is a product of METTLER TOLEDO, a reputable German company. It offers an impressive readability of 0.1mg and a substantial weight capacity, approximately 220g, as referenced in [42]. For comprehensive insights into the specific components and technical specifications of this balance, please consult the information provided in Appendix D. The balance's operation, illustrated in Figure 6.5, involves its integration with two robust metal ground plates and the inclusion of four standing rubber buffers, which collectively contribute to the system's ensured stability during the weighing process.

6.4 Ned2 Collaborative Robot

Ned2 is a collaborative robot, often referred to as a cobot, developed by the French company Niryo [10]. This particular cobot has been purpose-built for educational and research applications, serving as a valuable tool for the development of proof of concepts and experimental work. In the context of this research, the Ned2 cobot played a pivotal role in conducting experiments.

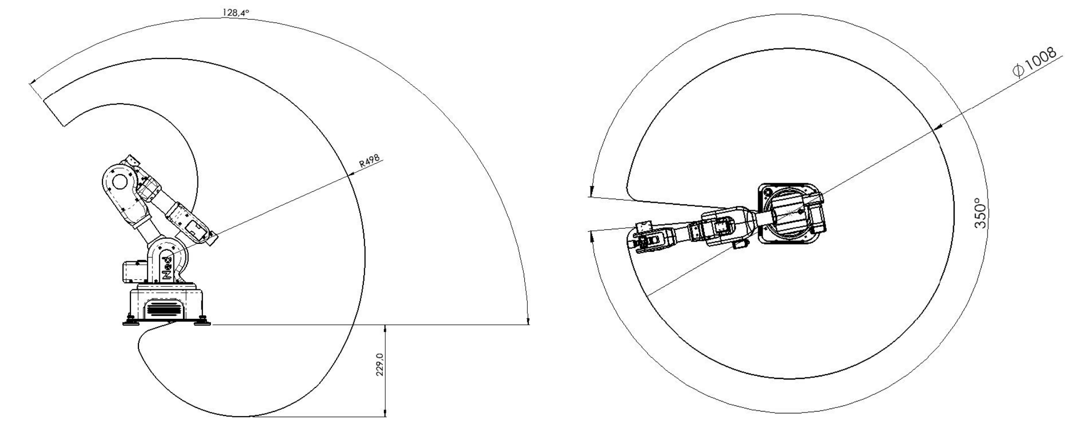
The forthcoming sections delve into an in-depth examination of the hardware specifications and software options offered by the Ned2 cobot.

FIGURE 6.6: Ned2 cobot provided by Niryo [10].

6.4.1 Hardware Configurations

Ned2 is a six-axis collaborative robot, based on open-source technologies. It is intended for education, research and Industry 4.0." [9]

Incorporating the same aluminum framework as its predecessor, Ned2 maintains its commitment to meeting your exacting standards in terms of durability, precision, and repeatability (with an accuracy, and a repeatability of 0.5 mm).



(a) (b)

FIGURE 6.8: The detailed workspace of Ned2 cobot.

Ned2 operates on the Ubuntu 18.04 platform and utilizes the ROS Melodic framework, capitalizing on the capabilities of the **Raspberry Pi 4**. This high-performance **64-bit ARM V8 processor**, coupled with **4GB of RAM**, empowers Ned2 to deliver enhanced performance.

(a) 2x XL430 (b) 1x XL330

FIGURE 6.7: The Servo motors used in Ned2.

This iteration of Ned2 introduces advanced servo motors equipped with Silent Stepper Technology, significantly reducing the operational noise of the robot. As a six-axis robot, the device comprises a total of six distinct motors. The initial three motors are specifically customized and operate as silent Stepper motors, delivering 65 $N - cm$ stall torque. These are equipped with a custom-made control card developed by Niryo. The remaining three motors are servos, with two being XL430 type motors delivering 140 $N - cm$ stall torque, and one XL330 type motor providing 65 $N - cm$ stall torque, as shown in Figure 6.7 [9]. The technical specifications of Ned2 are described as shown in table 6.1 below.

Furthermore, it's crucial for each robot to operate within a well-defined workspace that allows for unrestricted movement. In our case, the workspace of Ned2, as depicted in Figure 5.4 (a) and (b), has been meticulously documented in accordance with Niryo company specifications [9]. Our frame was meticulously designed and engineered to align with the defined workspace of the cobot.

Parameters	Value
Weight (kg)	7
Payload (g)	300
Reach (mm)	440
Degree of freedom	6 rotating joints
Joints range (rad)	$2,949 \leq Joint1 \leq 2,949$ $-2,09 \leq Joint2 \leq 0,61$ $-1.34 \leq Joint3 \leq 1,57$ $-2,089 \leq Joint4 \leq 2,089$ $-1,919 \leq Joint5 \leq 1.922$ $-2,53 \leq Joint6 \leq -2,53$
Joints speed limit (rad/s)	$Joint1 \leq 0.785$ $Joint2 \leq 0.5235$ $Joint3 \leq 0.785$ $Joint4 \leq 1.57$ $Joint5 \leq 1.57$ $Joint6 \leq 1.775$
TCP max speed (mm/s)	468
Repeatability (mm)	+/- 0,5
Footprint (mm)	200x200
Power supply	Input: AC100-240V / 50-60Hz, 2,5A Output: DC 12V - 7A ; 5V - 7A
I/O power supply	5V
Inputs/Outputs Control panel	Digital input x1 Digital output x1
Robot interface	USB2.0 x2 USB3.0 x2 ETHERNET GIGABIT x1
Communication	Modbus TCP (master) TCP/IP
Materials	Aluminum ABS-PC (injection moulding)
Collision detection	Accelerometer & gyroscope in the control panel
Certification	CE Conformity

TABLE 6.1: Technical Specification of Ned2 [9]

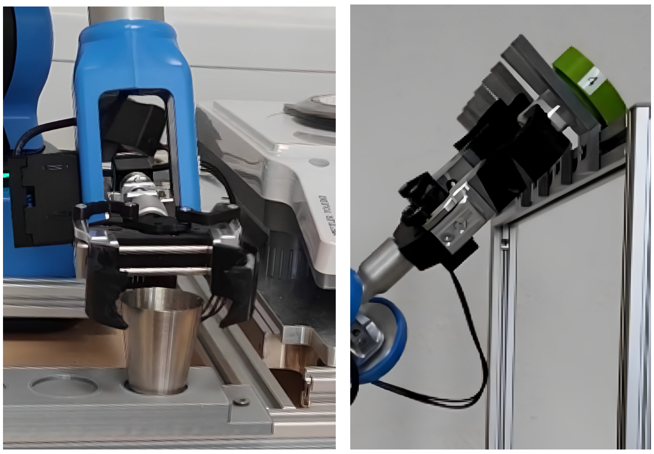
Cobot's Gripper

(a) The Initial phase of developing a gripper for the metal crucible only.

(b) The final phase of developing the gripper for both crucibles used in the research.

FIGURE 6.9: The development phase of the gripper.

While it's common to use general-purpose grippers such as those available from Niryo company, our research and development application necessitated a customized gripper. We have engineered a specialized gripper tailored to the unique demands of our project. In Figure 6.9, we can see the development phase of the gripper in order to fit the application for both glass and metal crucibles. The main tasks that the gripper is going to be used are holding the metal crucible from the middle, as shown in part (a) Figure 6.10, and holding the glass crucible from the bottom by the gripper's tip, as shown in part (b) Figure 6.10



(a) The initial task.

(b) The second task.

FIGURE 6.10: The main tasks of the gripper.

6.4.2 Software Tools

Ned2 represents a collaborative robot, hinging on the Ubuntu 18.04 platform and ROS Melodic—a widely adopted open-source solution in the field of robotics. Leveraging ROS, Ned2 offers an extensive array of libraries that empower users to create a wide spectrum of programs, from the simplest to the most intricate, thus ensuring adaptability to diverse operational requirements [9].

FIGURE 6.11: Software tools supported by Niryo in Ned2. [38]

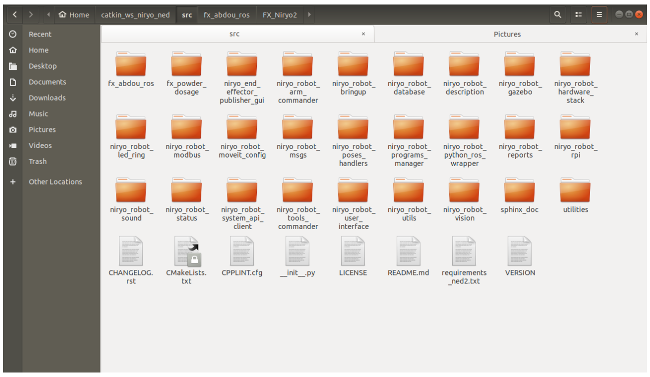


FIGURE 6.12: The Catkin workspace in Ned2

The software tools available for connecting and controlling the Niryo Ned2 cobot are extensive and diverse. They primarily include Niryo Studio, an interface offering multiple connectivity options such as hotspot, Wi-Fi, and Ethernet modes, each prompting calibration. This interface allows users to configure workspaces, adjust camera parameters, and control the conveyor belt. The cobot operates on ROS, featuring a range of proprietary and ready-made packages available on GitHub. It integrates motion planning capabilities using the MoveIt package, particularly employing the KDL kinematics plugin [33]. Thanks to Tensorflow (open-source machine learning tool developed by Google) [40], and OpenCV (open-source computer vision and machine learning software library) [39] training a computer vision-based model and recognizing objects is possible with Ned2. It supports multiple programming languages, including Blockly, Python, MATLAB [4], and interfaces with Arduino controllers [37], Modbus TCP servers, and various other languages via a TCP/IP server. Moreover, it provides the convenience of running Python scripts directly on the robot using the Python ROS wrapper (PyNiryo), making code execution simpler. Extensive tutorials, online documentation [9], and a dedicated support team is available for in-depth exploration and assistance. [11]
However, This research is conducting using ROS-Melodic version, and communication is done through ROS-master communication, with mainly python scripts.

ROS-Melodic

Since the ROS Melodic version is installed and used in the raspberry in Ned2 cobot, we had to use Ubuntu 18.4 on the PC where we connected with the robot through ROS-Master communication. Please refer to section 3.1 for more info about Linux.
Niryo has provided various ROS packages in their workspace, as shown in Figure 6.12 below.

6.5 Precision Validation

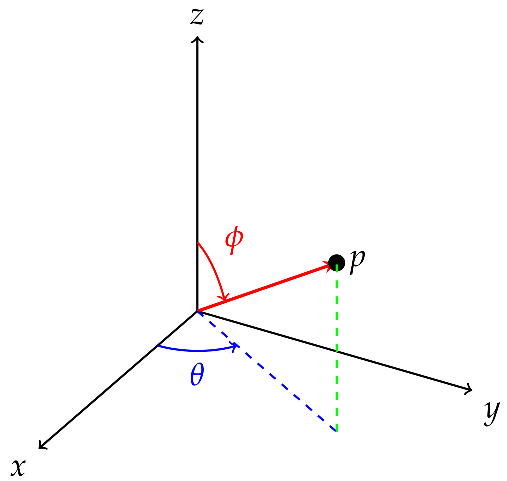
6.6 Vibration Motor

Chapter 7

Evaluation & Results

7.1 Evaluation

This is just a box!



Chapter 8

Conclusion & Future Work

small description of what I have done.. What are my final findings and thoughts...

The future work, what to come.

8.1 Future Work

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int32
rospy.init_node('topic_publisher')
pub = rospy.Publisher('counter', Int32)
rate = rospy.Rate(2)
count = 0

while not rospy.is_shutdown():
    pub.publish(count)
    count += 1
    rate.sleep()
```

LISTING 8.1: Some Python code

Appendix A

Frequently Asked Questions

Appendix B

FX_ROS.py Library in Python

```python
#!/usr/bin/env python
import tf
import time
import sys

# ROS client API
import rospy

# ROS services
from niryo_robot_arm_commander.srv import GetFK, GetFKRequest,
    GetJointLimits, JogShift, JogShiftRequest, JogShiftResponse
from moveit_msgs.srv import GetPositionFK

# ROS messages
from sensor_msgs.msg import JointState
from std_msgs.msg import Header
from moveit_msgs.msg import RobotState as RobotStateMoveIt
from geometry_msgs.msg import Pose
from niryo_robot_msgs.msg import RobotState
import moveit_msgs.msg

import moveit_commander
import actionlib

#robot = moveit_commander.RobotCommander()
#scene = moveit_commander.PlanningSceneInterface()
#global arm
def Connect_to_arm():
    global arm
    try:
        arm = moveit_commander.move_group.MoveGroupCommander("arm")
    except:
        raise RuntimeError

def Call_Aservice(service_name, type, request_name=None, req_args=None)
    :
    """Call a ROS service.

    Parameters:
    ........................
    service_name: str
    type: srv
    request_name: None (srv)
    req_args: None (dictionary) ex. {'positon': 210, 'id': 11, 'value':
     False}
    should_return ?: None (int) >> is set to 1, if you want to return
   the response of the service.

    Returns:
```

```python
    ........................
    If should_return is set to 1, the function is going to return the response of the service.
    Otherwise, the function should only call the service to do a certain action with no return.
    """
    try:
        rospy.wait_for_service(service_name, 2)
    except (rospy.ServiceException, rospy.ROSException) as e:
        rospy.logerr("Timeout and the Service was not available : " + str(e))
        return RobotState()

    try:
        service_call = rospy.ServiceProxy(service_name, type)

        if request_name == None:
            response = service_call()
        else:
            request = request_name()
            for key, value in req_args.items():
                #print("f{key} = {value}")
                method = setattr(request, key, value)
            response = service_call(request)

    except rospy.ServiceException as e:
        rospy.logerr("Falied to call the Service: " + str(e))
        return 0

    return response

def Subscribe(topic_name, type, msg_args):
    """Subscribe to a certain topic.

    Parameters:
    ........................
    topic_name: str
    type: srv
    msg_args: list >> list of strings, which contains the arguments that we need to read from the topic.

    Returns:
    ........................
    Return a list of the read values from each argument.
    If we have only one argument, it returns the value of this argument only, not a list.
    """

    #rospy.init_node('FX_ROS_Subscriber')

    try:
        msg = rospy.wait_for_message(topic_name, type, 2)
    except:
        rospy.logerr("Timeout and the Topic Did not recieve any messages")
        return 0


    value = []

    if len(msg_args) == 1:
        value = getattr(msg, msg_args[0])
    else:
```

```
        for i in msg_args:
            value.append(getattr(msg, i))

    return value

def Get_joints():
    """return a tuple of 6 values for each joint from 1 till 6"""

    joints_values = Subscribe('/joint_states', JointState, ["position"])

    return joints_values

def get_pose():
    """Gets the pose values from the robot_state topic.
    Return:
    ..................................
    a list of two dictionaries, the first is positions (x,y,z),
    whereas the second is the rpy (roll, pitch, yaw)
    """

    return Subscribe('/niryo_robot/robot_state', RobotState, ['position', 'rpy'])

def get_pose_list():
    """Use get_pose() function to get the pose, and turn it into a list.
    Return:
    ............................
    A list of floats >>> [x, y, z, roll, pitch, yaw]
    """

    pose = get_pose()
    position = pose[0]
    rpy = pose[1]

    return [position.x, position.y, position.z, rpy.roll, rpy.pitch, rpy.yaw]

def Get_FK_Niryo(joints):
    """Give the the joints' values to the forward kinematics service
    provided by Niryo, and get the pose coordinations.
    """
    fk_service = '/niryo_robot/kinematics/forward'
    return Call_Aservice(fk_service, GetFK, GetFKRequest, {'joints': joints}, should_return=1).pose

def FK_Moveit(joints):
    """Get Forward Kinematics from the MoveIt service directly after giving joints
    :param  joints
    :type   joints: list of joints values
    :return: A Pose state object
    @example of a return

position:
  x: 0.278076372862
  y: 0.101870353599
  z: 0.425462888681
orientation:
  x: 0.0257527874589
  y: 0.0122083384395
  z: 0.175399274203
```

```
  w: 0.984084775322

    """
    rospy.wait_for_service('compute_fk', 2)
    moveit_fk = rospy.ServiceProxy('compute_fk', GetPositionFK)

    fk_link = ['base_link', 'tool_link']
    header = Header(0, rospy.Time.now(), "world")
    rs = RobotStateMoveIt()

    rs.joint_state.name = ['joint_1', 'joint_2', 'joint_3', 'joint_4', 'joint_5', 'joint_6']
    rs.joint_state.position = joints

    reponse = moveit_fk(header, fk_link, rs)

    return reponse.pose_stamped[1].pose


def Jog_shift(joints_or_pose, axis, value):
    """Use the service jog_shift_commander to shift one axis.
    Paramters:
    ........................
    joints_or_pose: int >>> 1 for joints_shift, and 2 for pose_shift
    axis: int >>> (1,2,3,4,5,6) = (x,y,z,roll,pitch,yaw)
    value: float >> the value for which you want to shift the Jog axis.

    Returns: None
    ........................
    """

    axis -= 1
    name = "/niryo_robot/jog_interface/jog_shift_commander"
    shift_values = [0, 0, 0, 0, 0, 0]
    shift_values[axis] = value

    req_arg = {'cmd': joints_or_pose, 'shift_values': shift_values}

    Call_Aservice(name, JogShift, JogShiftRequest, req_arg)

def Move_pose_axis(axis, new=None, add=None, arm_speed=None):
    """You should either put a value to add or new, not both.

    Parameters:
    ........................
    * axis: str -> (x, y, z, roll, pitch, or yaw)
    * new: float -> The new coordination you want to give to a certain axis.
           "new" will always overwrite the value of the axis.
    * add: float -> the value in meters or radians you want to add to a certain axis.
    * arm_speed: float (optional) -> between 0 and 1. (0,1]
    Returns: None
    ........................
    """
    FK = get_pose()
    axises = ['x','y','z']

    pose = Pose()
    p_goal = pose.position
    orn_goal = pose.orientation

    p_current = FK[0]
```

```

    rpy_current = FK[1]

    if add:
        if axis.lower() in axises:
            current_value = getattr(p_current, axis)
            setattr(p_current, axis, current_value+add)
        else:
            current_value = getattr(rpy_current, axis)
            setattr(rpy_current, axis, current_value+add)
    if new:
        if axis.lower() in axises:
            setattr(p_current, axis, new)
        else:
            setattr(rpy_current, axis, new)


    p_goal.x = p_current.x
    p_goal.y = p_current.y
    p_goal.z = p_current.z

    orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
    .quaternion_from_euler(rpy_current.roll,rpy_current.pitch,
    rpy_current.yaw)

    arm.set_pose_target(pose)

    if arm_speed:
        set_speed(arm_speed)
    arm.go(wait=True)

    arm.stop()
    arm.clear_pose_targets()

def Move_to_pose(pose_values, arm_speed=None):
    """Move to a given pose values.
    Parameters:
    ........................

    pose_values: list or tuble -> [x, y, z, roll, pitch, yaw]
    arm_speed: float (optional) -> between 0 and 1. (0,1]
    """

    pose = Pose()
    p_goal = pose.position
    orn_goal = pose.orientation

    p_goal.x = pose_values[0]
    p_goal.y = pose_values[1]
    p_goal.z = pose_values[2]

    roll = pose_values[3]
    pitch = pose_values[4]
    yaw   = pose_values[5]

    orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
    .quaternion_from_euler(roll,pitch,yaw)

    #arm.set_goal_tolerance(0.001)
    if arm_speed:
        set_speed(arm_speed)
    arm.set_pose_target(pose)
    arm.go(wait=True)
```

```python

    arm.stop()
    arm.clear_pose_targets()

def move_to_joints(joints, arm_speed=None):
    """Move to a given joint values.
    Parameters:
    ........................

    joints: list or tuble -> [joint1, joint2, joint3, joint4, joint5, joint6]
    arm_speed: float (optional) -> between 0 and 1. (0,1]
    """
    joints_limits = Get_Joints_limits()

    for i in range(6):
        if joints_limits.joint_limits[i].max < joints[i] or joints[i] < joints_limits.joint_limits[i].min:
            print("Joint{} = {}, which is out of limit!".format(i+1, joints[i]))
            print("Joint{} can not be more than {} neither less than {}".format(i+1, joints_limits.joint_limits[i].max, joints_limits.joint_limits[i].min))
            return
        else:
            pass

    #arm.set_joint_value_target(joints)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(joints, wait=True)

    arm.stop()

def Move_joint_axis(axis, new=None, add=None, arm_speed=None):
    """You should either put a value to add or new, not both.

    Parameters:
    ........................
    * axis: int -> the number of the joint that you want to move

    * new: float -> The new coordination you want to give to a joint (axis).
           "new" will always overright the value of the axis.
    * add: float -> the value in meters change in a certain joint (axis).
    * arm_speed: float (optional) -> between 0 and 1. (0,1]

    Returns: None
    ........................
    """
    moving_joints = list(Get_joints())

    if new:
        moving_joints[axis-1] = new
    elif add:
        moving_joints[axis-1] += add

    joints_limits = Get_Joints_limits()

    if joints_limits.joint_limits[axis-1].max < moving_joints[axis-1] or moving_joints[axis-1] < joints_limits.joint_limits[axis-1].min:
```

```
        print("The joint{} can not be more than {} neither less than {}".format(axis, joints_limits.joint_limits[axis-1].max, joints_limits.joint_limits[axis-1].min))
        return 0
    else:
        pass

    arm.set_joint_value_target(moving_joints)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(moving_joints, wait=True)

    arm.stop()

def Get_Joints_limits():
    """Getting the limits for each joint.

    You can get any joint limits as following:

    Get_Joints_limits().joint_limits[0 - 5].max (float)
    Get_Joints_limits().joint_limits[0 - 5].min (float)
    Get_Joints_limits().joint_limits[0 - 5].name (str)

    Where 0 for (joint 1), and 5 for (joint 6)
    max, min, or name would give the maximum, minimum, or name of the indicated joint.
    """

    joints_limits = Call_Aservice('/niryo_robot_arm_commander/get_joints_limit', GetJointLimits)
    return joints_limits

def set_speed(speed):
    """Set a scaling factor for optionally reducing the maximum joint velocity. Allowed values are in (0,1]."""
    arm.set_max_velocity_scaling_factor(speed)

def wait(duration):
    """wait for a certain time.

    :param duration: duration in seconds
    :type duration: float
    :rtype: None
    """
    time.sleep(duration)

def move_with_action(pose):
    """Still under development"""

    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('simple_action', anonymous=True)

    robot_arm = moveit_commander.move_group.MoveGroupCommander("arm")

    robot_client = actionlib.SimpleActionClient('execute_trajectory', moveit_msgs.msg.ExecuteTrajectoryAction)
    robot_client.wait_for_server()
    #rospy.loginfo('Execute Trajectory server is available for robot')

    robot_arm.set_pose_target(pose)
    #robot_arm.set_pose_target([0.29537095654868956, 4.675568598554573e-05, 0.4286678926923855, 0.0017192879795506913, 0.0014037282477544944, 0.00016120358136762693])
```

```python
    robot_plan_home = robot_arm.plan()

    robot_goal = moveit_msgs.msg.ExecuteTrajectoryGoal()
    robot_goal.trajectory = robot_plan_home

    robot_client.send_goal(robot_goal)
    robot_client.wait_for_result()
    robot_arm.stop()

def move_pose_orn(pose, arm_speed=None):
    """Move to a given pose values, but with orientation not rpy.

    Parameters:
    ........................
    * pose: A Pose state object

example of the pose state object that should be given:
    ========================
    position:
        x: 0.278076372862
        y: 0.101870353599
        z: 0.425462888681
    orientation:
        x: 0.0257527874589
        y: 0.0122083384395
        z: 0.175399274203
        w: 0.984084775322
    ========================
    """

    arm.set_pose_target(pose)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(wait=True)

    arm.stop()
    arm.clear_pose_targets()

def move_to_named_pos(position_name, arm_speed=None):
    """Avalible names:
    - 'resting'
    - 'straight_forward'
    - 'straight_up'
    """
    arm.set_named_target(position_name)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(wait=True)
```

Appendix C

xlsx.py Package in Python

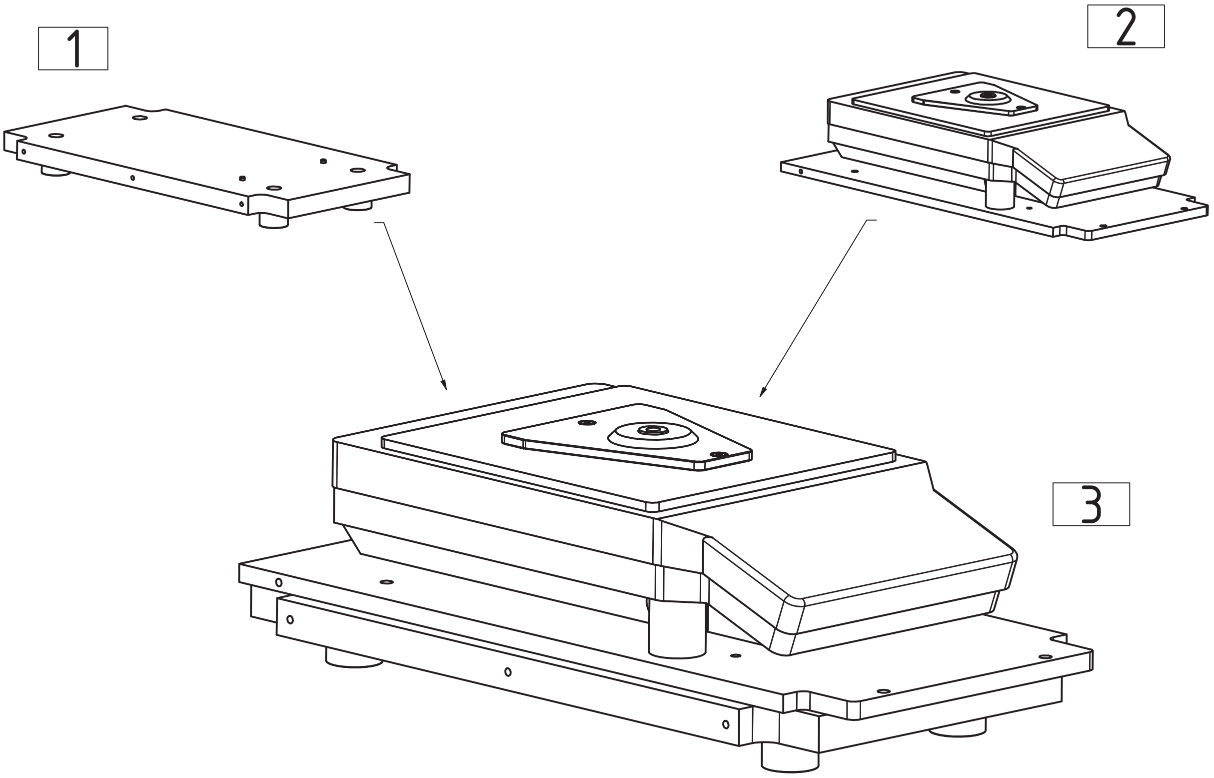
```
from openpyxl import Workbook, load_workbook
from openpyxl.utils import get_column_letter
from openpyxl.styles import Font, Alignment

positions_dict = {}
headers = ['Name', 'Joint_values', 'Pose_values']

def Creat_worksheet(document_name='Untitled'):
    global positions_dict

    if ".xlsx" != document_name[-5:]:
        document_name += '.xlsx'

    # Create a workbook object
    wb = Workbook()

    # Create an active worksheet
    ws = wb.active
    ws.title = 'Data'

    for col in range(1, 4):
        ws[get_column_letter(col) + '1'] = headers [col-1]
        ws[get_column_letter(col) + '1'].font = Font(bold=True, color="
    0099CCFF")
        ws[get_column_letter(col) + '1'].alignment = Alignment(
    horizontal='center')

    ws.move_range("C1", rows=0, cols=5)
    ws.column_dimensions['A'].width = 20

    ws.merge_cells("B1:G1")
    ws.merge_cells('H1:M1')

    wb.save(document_name)
    wb.close()

def Save_position(document_name, position_name, joint_values,
    pose_values):
    """Paramters:
    ........................
    document_name: str
    position name: str
    joint_values: list >> list of 6 floats.
    pose_values: list >> list of 6 floats.

    Returns:
    ........................
    """
    global positions_dict
```

```

    if ".xlsx" != document_name[-5:]:
        document_name += '.xlsx'

    wb = load_workbook(document_name)
    ws = wb.active

    ws.append([position_name] + joint_values + pose_values)
    wb.save(document_name)
    wb.close()

def Load_all_positions(document_name):
    """Paramter:
    ........................
    document_name: str --> the name of the excel document you want to load the positions from

    Return:
    ........................
    position_dict: dictionary --> A dictionary which contains all the positions from the given excel file.
    The dict has the following structure:
    {'Position_name':{'Joint_values':[j1,j2,j3,j4,j5,j6], 'Pose_values':[x,y,z,roll,pitch,yaw]}}
    """
    global positions_dict

    if ".xlsx" != document_name[-5:]:
        document_name += '.xlsx'

    wb = load_workbook(document_name)
    ws = wb.active


    for row in ws.iter_rows(min_row=2, max_col=13):
        if row[0].value == None:
            ws.delete_rows(row[0].row)
            wb.save(document_name)
        positions_dict[row[0].value] = {}
        joints = []
        pose = []
        for i in range(1,7):
            joints.append(row[i].value)
            pose.append(row[i+6].value)
        positions_dict[row[0].value]['Joint_values'] = joints
        positions_dict[row[0].value]['Pose_values'] = pose

    wb.close()
    return positions_dict

def Load_position(document_name, position_name, debug=True):
    global positions_dict

    if ".xlsx" != document_name[-5:]:
        document_name += '.xlsx'

    wb = load_workbook(document_name)
    ws = wb.active

    for row in ws.iter_rows(min_row=2, min_col=1, max_col=13):
        if row[0].value == position_name:
            if position_name in positions_dict.keys():
```

```
                  if debug: print("The position is already loaded into positions dictionary")
                  wb.close()
                  return
            else:
                positions_dict[position_name] = {}
                joints = []
                pose = []
                for i in range(1,7):
                    #if row[i].value == None:
                        #break
                    joints.append(row[i].value)
                    pose.append(row[i+6].value)
                positions_dict[position_name]['Joint_values'] = joints
                positions_dict[position_name]['Pose_values'] = pose

                if debug: print('The position is successfully loaded into the positions dictionary')
                wb.close()
                return positions_dict

    print("Position was not found!")
    wb.close()

def delete_pos(document_name, position_name):
    if ".xlsx" != document_name[-5:]:
        document_name += '.xlsx'

    wb = load_workbook(document_name)
    ws = wb.active

    for row in ws.iter_rows(min_row=2, min_col=1, max_col=1):
        if position_name == row[0].value:
            ws.delete_rows(row[0].row)
            wb.save(document_name)
            wb.close()
            return

    wb.close()
    print("No position found with the name: '{}'".format(position_name))
```

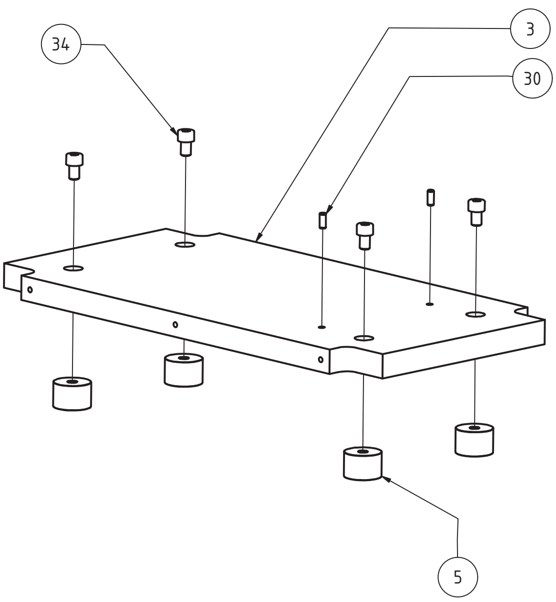
Appendix D

Balance Technical Drawing

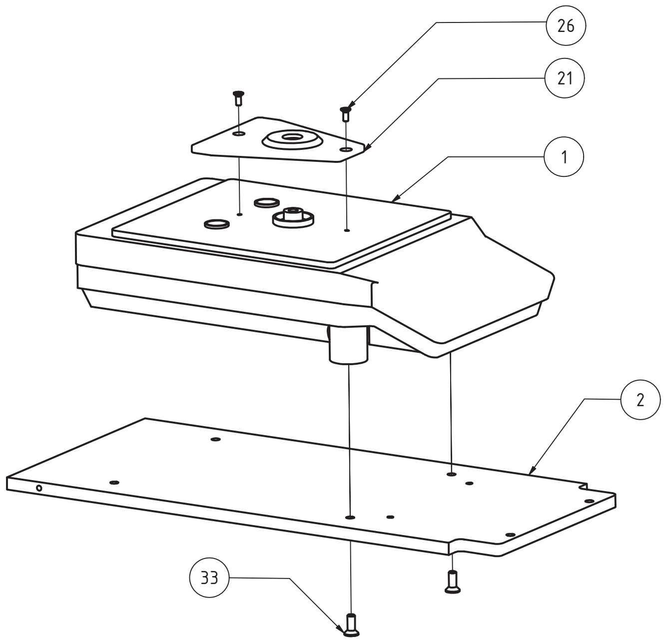


Fluxana GmbH & Co.KG		Datum	Name
	Gezeichn.	20.02.2019	J.Schramm
Bezeichnung: Schublade mit Waage	Kontroll.		
	Norm		
Projekt: Boramat 18 & 30			1 /3
Baugruppen-Nr. BG0106	Version: 2	Art.-Nr.	A3

Schritt 1: Grundplatte Schublade und Gummifüße

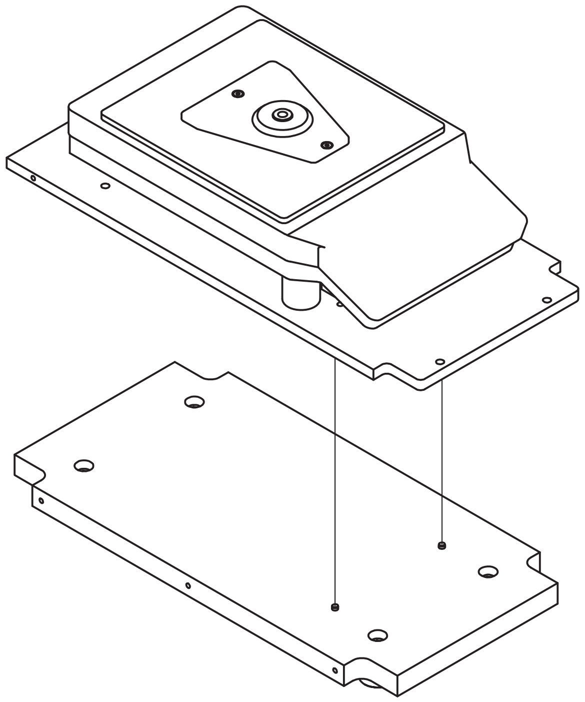


Schritt 2: Grundplatte Schublade und Analysenwaage



Fluxana GmbH & Co.KG			Datum	Name
Bezeichnung: Schublade mit Waage		Gezeichn.	20.02.2019	J.Schramm
		Kontroll.		
		Norm		
Projekt: Boramat 18 & 30				2 /3
Baugruppen-Nr. BG0106	Version: 2	Art.-Nr.		A3

Schritt 3: Montage der Führungsschienen

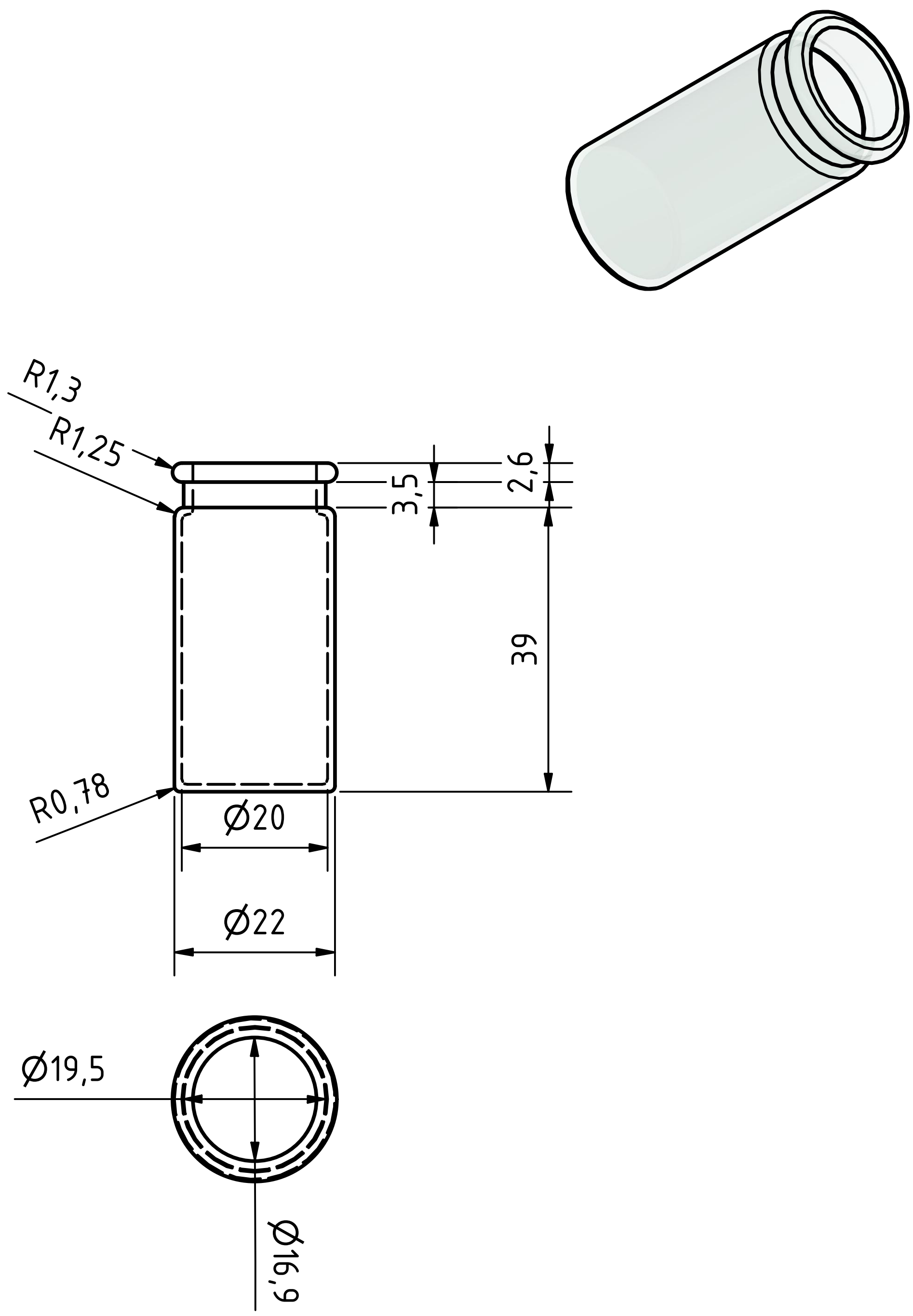


PARTS LIST			
ITEM	QTY	PART NUMBER	DESCRIPTION
1	1		Mettler Toledo Waage
2	1	BT0413	Grundplatte Waage
3	1	BT0414	Grundplatte Schublade
5	4		Gummipuffer 30x20mm M8x8
21	1	BT0419	Schutz Waage
26	2	DIN EN ISO 10642	Senkschraube mit Inbus M4 x 10
30	2	ISO 2338	Zylinderstifte 5 m6 x 12
33	2	DIN EN ISO 10642	Senkschraube mit Inbus M6 x 16
34	4	DIN EN ISO 4762	Zylinderschraube mit Inbus M8 x 12

Fluxana GmbH & Co.KG		Datum	Name
	Gezeichn.	20.02.2019	J.Schramm
Bezeichnung: Schublade mit Waage	Kontroll.		
	Norm		
Projekt: Boramat 18 & 30			3 /3
Baugruppen-Nr. BG0106	Version: 2	Art.-Nr.	A3

Appendix E

Glass and Metal Crucibles Technical Drawing



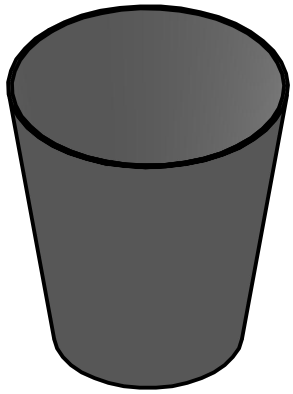
Allgemeintoleranz: DIN ISO 2768-mK			Alle Kanten R0,1 - 0,3 entgratet	
Erstellt durch: Abdelrahman Mostafa	Genehmigt von:	Gewicht: N/A		Werkstoff: Glas

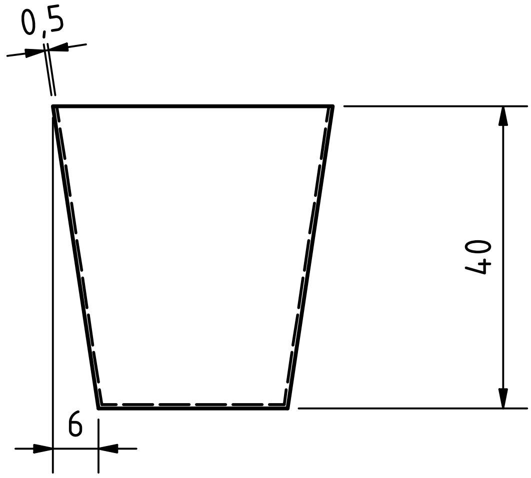
Bauteilbezeichnung:

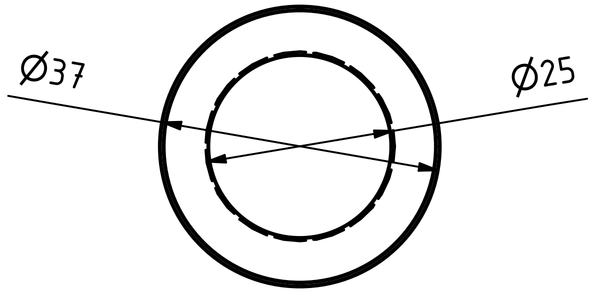
FLUXANA® XRF Application Solutions

Projekt:	Artikelnr.:
	Bauteilzeichnungsnr.: Schnappdeckelglas-1

Format	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4	1 : 1	0	24/10/2023	de	1/1







Allgemeintoleranz: DIN ISO 2768-mK			Alle Kanten R0,1 - 0,3 entgratet	
Erstellt durch: Abdelrahman Mostafa	Genehmigt von:	Gewicht: N/A		Werkstoff: Generisch
Bauteilbezeichnung:				

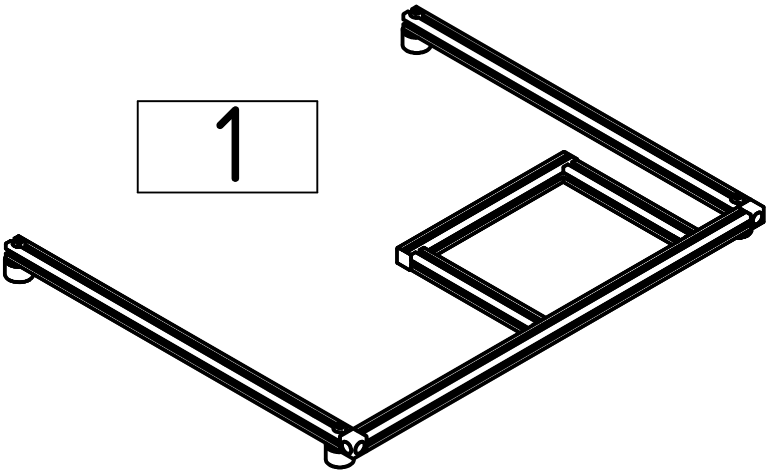
FLUXANA®
XRF Application Solutions

Projekt:	Artikelnr.:					
	Bauteilzeichnungsnr.: Boromat_tiegel-1					
	Format: A4	Maßstab: 1 : 1	Änd.: 0	Datum: 24/10/2023	Spr.: de	Blatt: 1/1

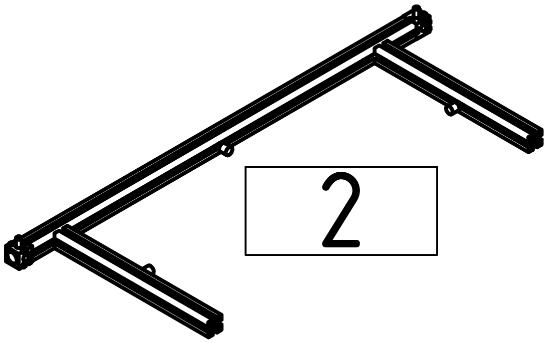
Appendix F

Frame Technical Drawing

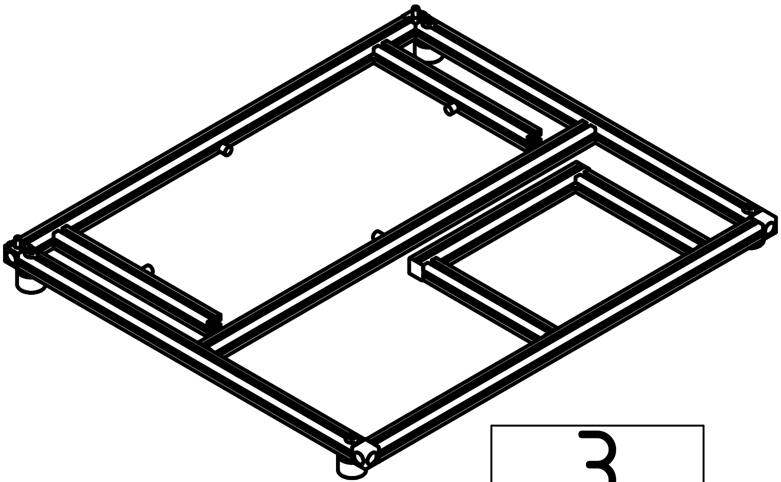
1



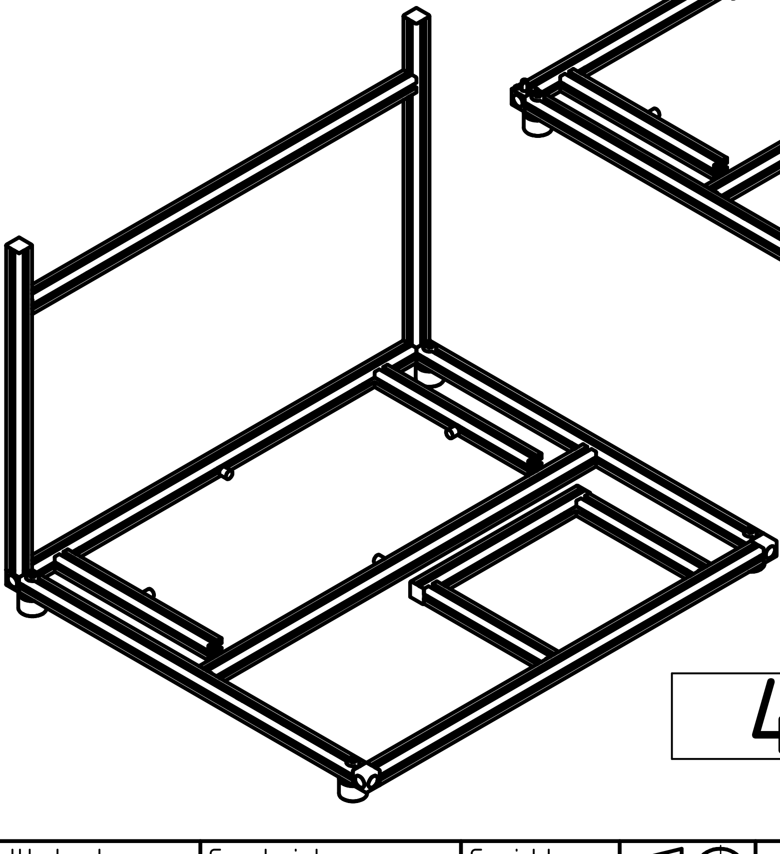
2



3



4



Erstellt durch:	Genehmigt von:	Gewicht:
Carine Allen		N/A

Montagebaugruppenbez.: Niryo Frame

FLUXANA® XRF Application Solutions

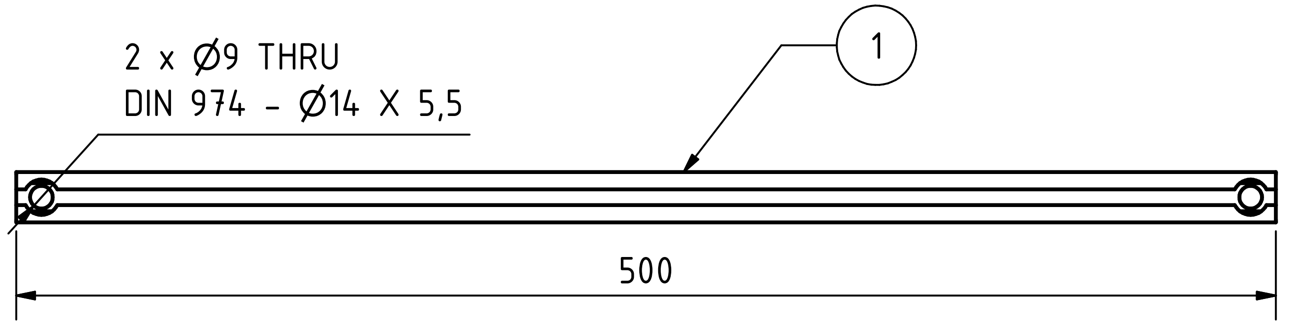
Projekt:

Artikelnr.:

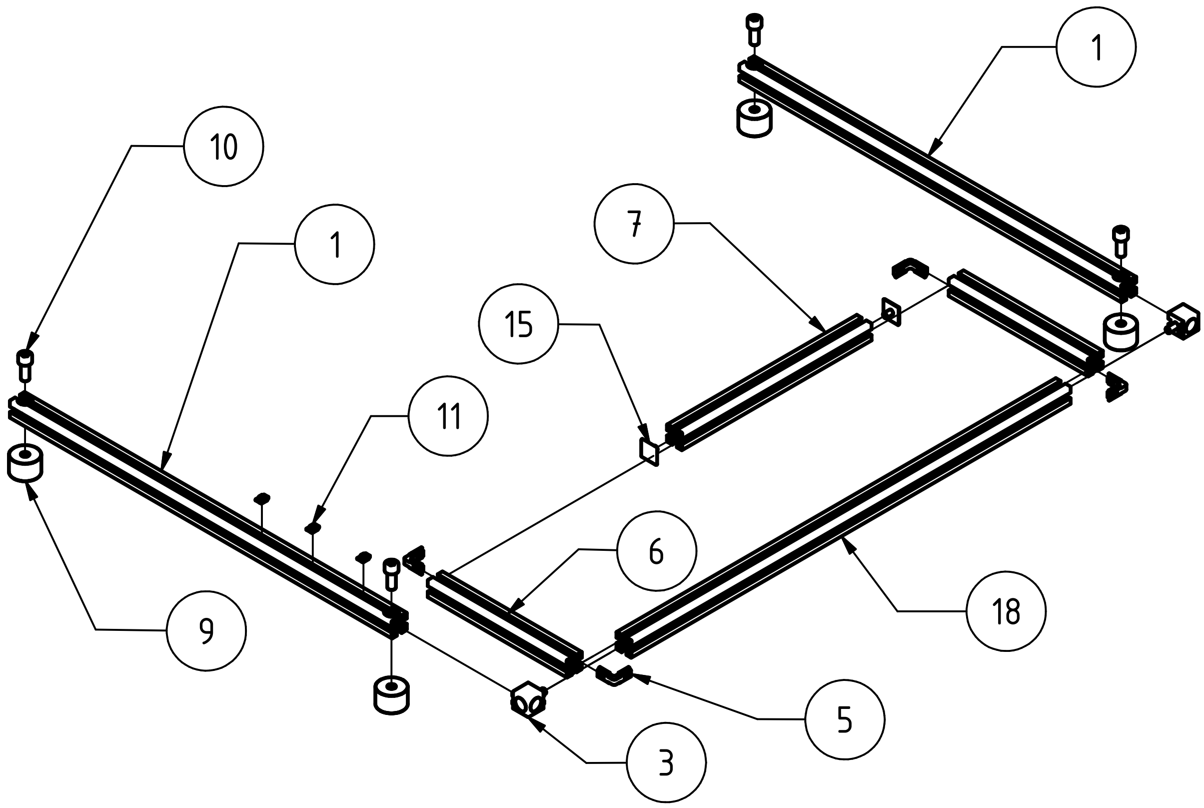
Montagebaugruppenzeichnungsnr.:

Format:	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4	1 : 8	0	21/06/2023	de	1/5

Punkt 1: 2 x Bohrung im Profil 20x200x500mm



Schritt 1: Niryo holder



Erstellt durch:	Genehmigt von:	Gewicht:
Carine Allen		N/A

Montagebaugruppenbez.: Niryo Frame

FLUXANA® XRF Application Solutions

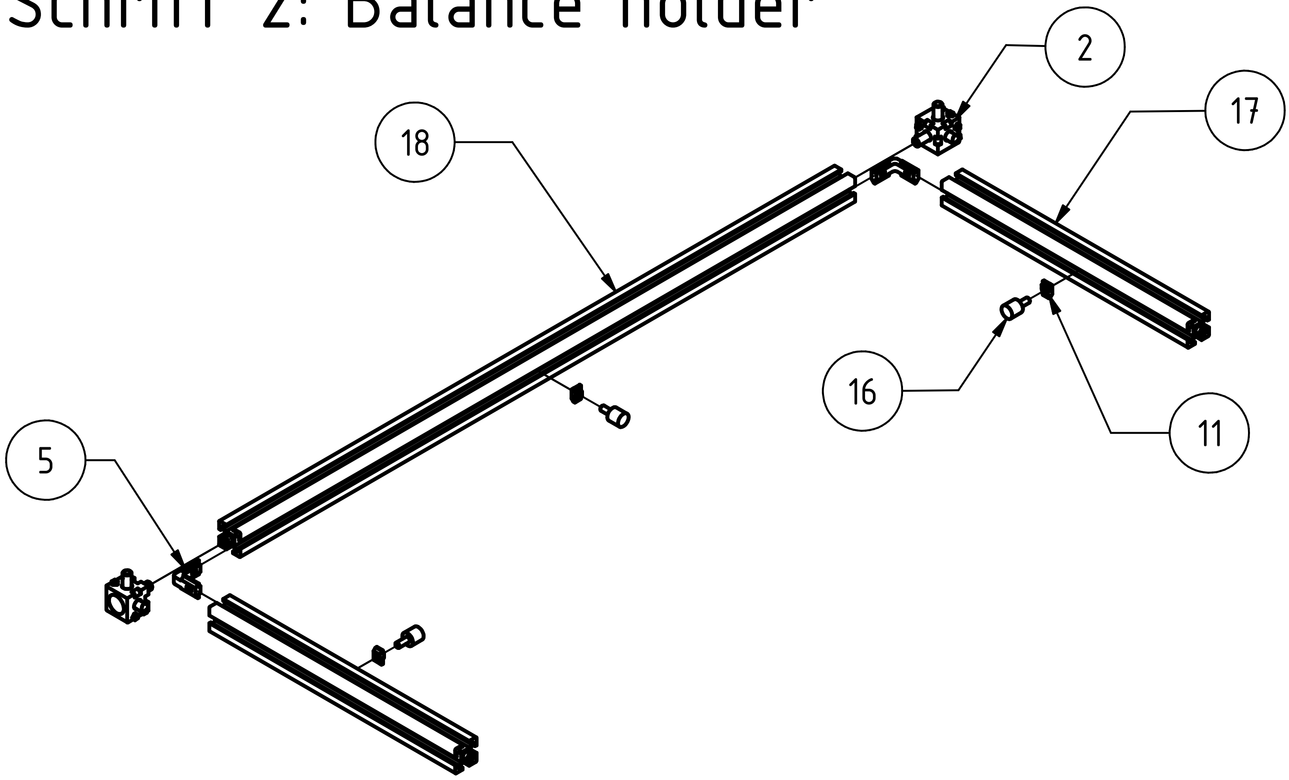
Projekt:

Artikelnr.:

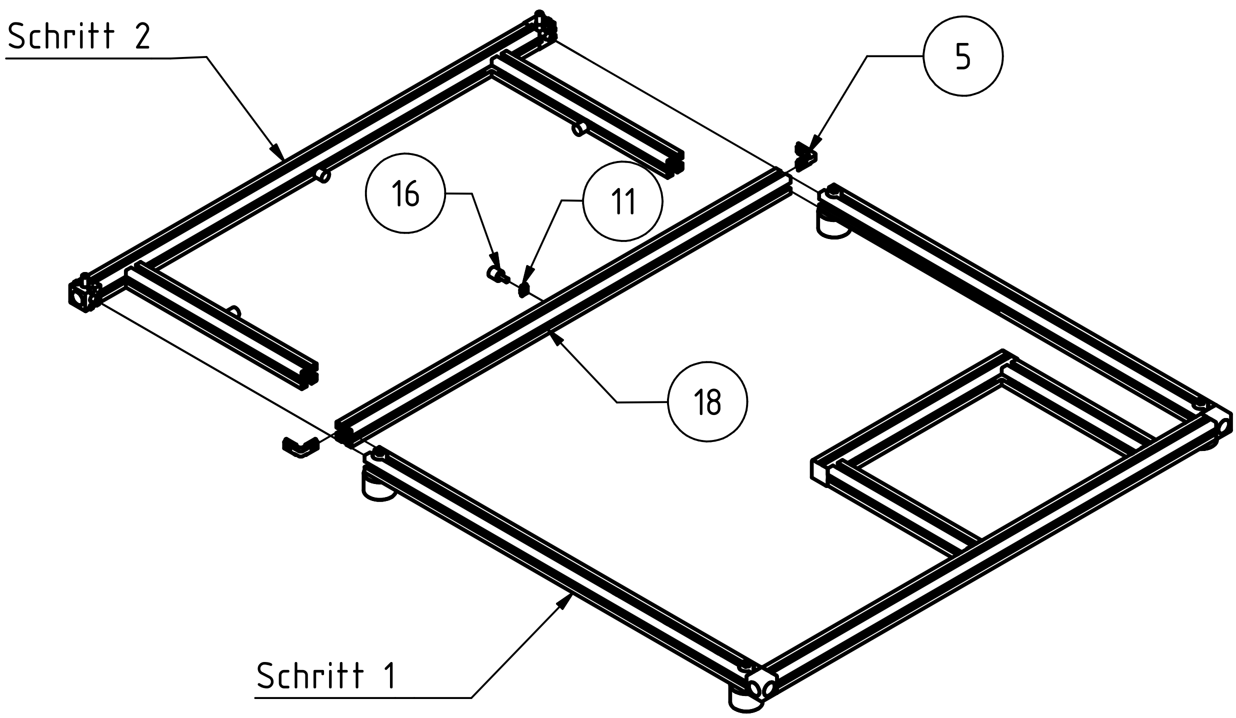
Montagebaugruppenzeichnungsnr.:

Format:	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4	1 : 7	0	21/06/2023	de	2/5

Schritt 2: Balance holder



Schritt 3: Base frame



Erstellt durch: Carine Allen	Genehmigt von:	Gewicht: N/A	

Montagebaugruppenbez.: Niryo Frame

FLUXANA®
XRF Application Solutions

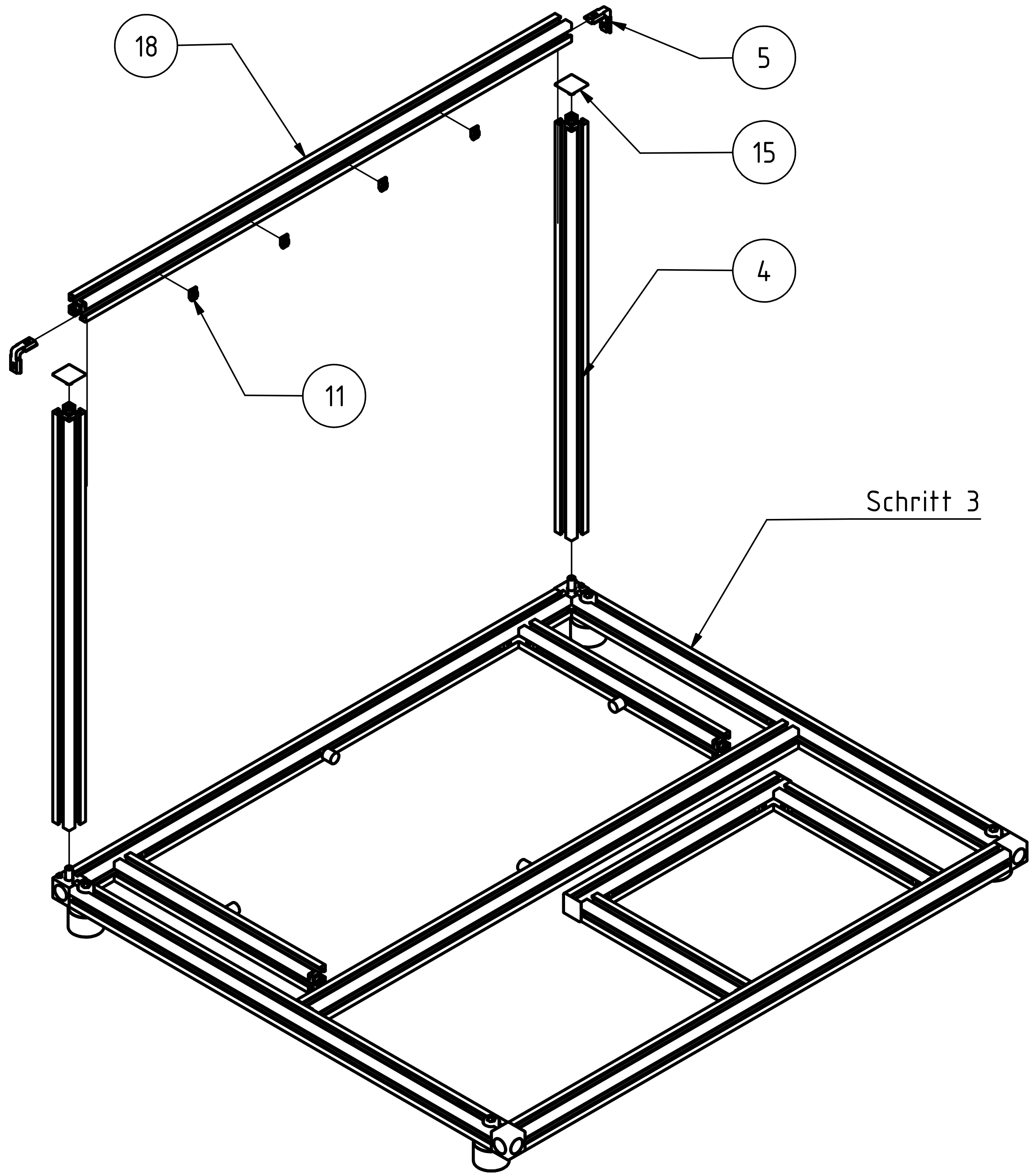
Projekt:

Artikelnr.:

Montagebaugruppenzeichnungsnr.:

Format:	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4	1 : 5	0	21/06/2023	de	3/5

Schritt 4: Complete frame



Erstellt durch: Carine Allen	Genehmigt von:	Gewicht: N/A	

Montagebaugruppenbez.: Niryo Frame

FLUXANA®
XRF Application Solutions

Projekt:

Artikelnr.:

Montagebaugruppenzeichnungsnr.:

Format:	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4	1 : 5	0	21/06/2023	de	4/5

Montagebaugruppenstückliste

Pos.	Anz.	Artikelnr.	Norm-/ BT-nr	Beschreibung
1	2	VI-0342		Strebenprofil 20x20x500mm
2	2	VI-0348		Würfelverbinder 20/3
3	2	VI-0347		Würfelverbinder 20/2
4	2	VI-0342		Strebenprofil 20x20x420mm
5	10	VI-0344		Innenwinkel 6R
6	2	VI-0342		Strebenprofil 20x20x182mm
7	1	VI-0342		Strebenprofil 20x20x248mm
8	1		BG0106	Schublade mit Waage
9	4	BO-0222		Gummipuffer 30x20mm M8x8
10	4		ISO 4762 - M8 x 20	Hexagon Socket Head Cap Screw
11	11	KT-06-0012		Hammermutter 6 M4
12	4		Becherglass_hanger	
13	1		Schnappdeckelglas	
14	1	BO-0289	BT0953	Adapterring Schnappdeckelglas Boramat 30
15	4	KT-07-0001		Abdeckkappe grau 20x20
16	4	GR-0186		Gummipuffer D10 x H10 M4x10AES
17	2	VI-0342		Strebenprofil 20x20x230mm
18	4	VI-0342		Strebenprofil 20x20x575mm
19	3		Schnappdeckelglas_hanger	

Erstellt durch: Carine Allen	Genehmigt von:	Gewicht:

Montagebaugruppenbez.: Niryo Frame

FLUXANA®
XRF Application Solutions

Projekt:

Artikelnr.:

Montagebaugruppenzeichnungsnr.:

Format:	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4		0	21/06/2023	de	5/5

Bibliography

[1] Robert O Ambrose et al. “Robonaut: NASA’s space humanoid”. In: *IEEE Intelligent Systems and Their Applications* 15.4 (2000), pp. 57–63.

[2] Arsalan Anwar. *simple steps to create and build your first ROS package*. https://medium.com/swlh/7-simple-steps-to-create-and-build-our-first-ros-package-7e3080d36faa. Published in. 2021,

[3] Tessa Brazda. *NASA Robonaut first generation*. https://www.nasa.gov/technology/r1-the-first-generation/. found in. 2023.

[4] Praneel Chand. “Developing a Matlab Controller for Niryo Ned Robot”. In: *2022 1st International Conference on Technology Innovation and Its Applications (ICTIIA)*. IEEE. 2022, pp. 1–5.

[5] David Coleman et al. “Reducing the barrier to entry of complex robotic software: a moveit! case study”. In: *arXiv preprint arXiv:1404.3785* (2014).

[6] Peter Corke. “Integrating ros and matlab [ros topics]”. In: *IEEE Robotics & Automation Magazine* 22.2 (2015), pp. 18–20.

[7] Anish Devasia. *PLC Sequencer Logic: An Overview*. https://control.com/technical-articles/plc-sequencer-logic-an-overview/. Aug. 2020.

[8] Farbod Fahimi. *Autonomous robots*. Springer, 2009.

[9] Marc-Henri Frouin. *Ned2 Documentation*. https://docs.niryo.com/product/ned2/v1.0.0/en/source/introduction.html. found in. 2022.

[10] Marc-Henri Frouin. *Niryo Company*. https://niryo.com. found in. 2017.

[11] Marc-Henri Frouin. *package Documentation*. https://docs.niryo.com/dev/ros/v4.1.1/en/source/stack/high_level.html. found in. 2022.

[12] Intel. *Types of Robots: How Robotics Technologies Are Shaping Today’s World*. https://www.intel.com/content/www/us/en/robotics/types-and-applications.html. found in. 2023.

[13] Reza N. Jazar. *Theory of Applied Robotics: Kinematics, Dynamics, and Control*. 3rd edition. Cham: Springer International Publishing, Imprint: Springer, 2022. DOI: 10.1007/978-3-030-93220-6.

[14] Lentin Joseph and Aleena Johny. “Getting Started with Ubuntu Linux for Robotics”. In: *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Springer, 2022, pp. 1–52.

[15] Lentin Joseph and Aleena Johny. “Programming with ROS”. In: *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Springer, 2022, pp. 173–240.

[16] Ltd. Kawasaki Heavy Industries. *Comparison between robot and human movement*. https://robotics.kawasaki.com/ja1/xyz/en/1804-03/. 2018.

[17] Charles C Kemp et al. “Humanoids”. In: *experimental psychology* 56 (2009), pp. 1–3.

[18] Jay LaCroix. *Mastering Ubuntu Server: Master the art of deploying, configuring, managing, and troubleshooting Ubuntu Server 18.04*. Packt Publishing Ltd, 2018.

[19] B.A. Lee M. "Robotics Background Information and Facts". In: *Salem Press Encyclopedia of Science* (2022). URL: https://bvyg.short.gy/robotics-background.

[20] Kevin M Lynch and Frank C Park. *Modern robotics*. Cambridge University Press, 2017. URL: www.cambridge.org/9781107156302.

[21] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. "On the benefits of making robotic software frameworks thin". In: *International Conference on Intelligent Robots and Systems-Workshop for Measures and Procedures for the Evaluation of Robot Architectures and Middleware at IROS'07*. Vol. 2. 2007.

[22] Janis Osis and Uldis Donins. "Chapter 1 - Unified Modeling Language: A Standard for Designing a Software". In: *Topological UML Modeling*. Ed. by Janis Osis and Uldis Donins. Computer Science Reviews and Trends. Boston: Elsevier, 2017, pp. 3–51. ISBN: 978-0-12-805476-5. DOI: https://doi.org/10.1016/B978-0-12-805476-5.00001-0. URL: https://www.sciencedirect.com/science/article/pii/B9780128054765000010.

[23] Jia Pan, Sachin Chitta, and Dinesh Manocha. "FCL: A general purpose library for collision and proximity queries". In: *2012 IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 3859–3866.

[24] Michael Peshkin and J Edward Colgate. "Cobots". In: *Industrial Robot: An International Journal* 26.5 (1999), pp. 335–341.

[25] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System*. " O'Reilly Media, Inc.", 2015.

[26] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. Kobe, Japan. 2009, p. 5.

[27] Open Robotics. *ROS*. https://www.ros.org/. found in. 2015.

[28] Open Robotics. *ROS-documantation*. https://docs.ros.org/. found in. 2015.

[29] Open Robotics. *ROS-Master*. https://wiki.ros.org/Master. found in. 2015.

[30] Open Robotics. *roscore*. https://wiki.ros.org/roscore. found in. 2015.

[31] Open Robotics. *rqt_graph*. https://wiki.ros.org/rqt_graph. found in. 2015.

[32] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[33] Ruben Smits, H Bruyninckx, and E Aertbeliën. *KDL: Kinematics and dynamics library*. https://www.orocos.org/kdl. found in. 2011.

[34] Andrew Spielberg et al. "Functional co-optimization of articulated robots". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 5035–5042.

[35] Adam A. Stokes et al. "A Hybrid Combining Hard and Soft Robots". In: *Soft Robotics* 1.1 (2014), pp. 70–74. DOI: 10.1089/soro.2013.0002.

[36] Ioan A. Sucan and Sachin Chitta. *MoveIt*. https://moveit.ros.org/. found in. 2013.

[37] Niryo Team. *Control Ned with an Arduino Board*. https://docs.niryo.com/applications/ned/v1.0.3/en/source/tutorials/control_ned_arduino.html. found in. 2022.

[38] Niryo Team. *Ned2 Overview*. https://niryo.com/products-cobots/robot-ned-2/. found in. 2022.

[39] Niryo Team. *PyNiryo OpenCV Example*. https://docs.niryo.com/dev/pyniryo/v1.1.2/en/source/vision/image_processing_overview.html. found in. 2023.

[40] Niryo Team. *Visual picking with Artificial Intelligence using TensorFlow*. https://niryo.com/learn-robotics/visual-picking-with-ai-tensor-flow/. found in. 2023.

[41] Inc. Teradyne. *Universal Robots*. https://www.universal-robots.com/. found in. 2005.

[42] METTLER TOLEDO. *METTLER TOLEDO ME BALANCE*. https://bvyg.short.gy/me_balance. 2020.

[43] Frantisek Trebuňa. *Applied mechanics and mechatronics*. Applied mechanics and materials ;; Applied mechanics and materials, ISSN 1662-7482. Trans Tech Publications, 2014. ISBN: 3-03826-572-1. URL: https://eu04.alma.exlibrisgroup.com/view/uresolver/49HBZ_RHW/openurl?u.ignore_date_coverage=true&portfolio_pid=5324821430006479&Force_direct=true.

[44] Günter Ullrich et al. "Automated guided vehicle systems". In: *Springer-Verlag Berlin Heidelberg. doi* 10 (2015), pp. 978–3.

[45] Keenan A Wyrobek et al. "Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot". In: *2008 IEEE International Conference on Robotics and Automation*. IEEE. 2008, pp. 2165–2170.